



Docket No.: 0039-7646-2RD

COMMISSIONER FOR PATENTS
ALEXANDRIA, VIRGINIA 22313



ATTORNEYS AT LAW

RE: Application Serial No.: 09/532,535
Applicants: Tatsunori KANAI, et al.
Filing Date: March 22, 2000
For: SCHEME FOR SYSTEMATICALLY REGISTERING
META-DATA WITH RESPECT TO VARIOUS
TYPES OF DATA
Group Art Unit: 2151
Examiner: F. JEAN

SIR:

Attached hereto for filing are the following papers:

Petition Under 37 C.F.R. § 1.181(A)(3) To Invoke The Supervisory Authority Of The
Commissioner, Copy of Filing Receipt Date-Stamped 03/22/00, Copy of Information Disclosure
Statement Filed 03/22/00, Copy of PTO-1449 Filed 03/22/00, Copy of Statement of Relevancy Filed
03/22/00, Copy of Filing Receipt Date-Stamped 04/30/02, Copy of Information Disclosure Statement
Filed 04/30/02, Copy of PTO-1449 Filed 04/30/02, Copy of Cited References (20)

Our check in the amount of \$0.00 is attached covering any required fees. In the event any variance exists between the amount enclosed and the Patent Office charges for filing the above-noted documents, including any fees required under 37 C.F.R. 1.136 for any necessary Extension of Time to make the filing of the attached documents timely, please charge or credit the difference to our Deposit Account No. 15-0030. Further, if these papers are not considered timely filed, then a petition is hereby made under 37 C.F.R. 1.136 for the necessary extension of time. A duplicate copy of this sheet is enclosed.

Respectfully submitted,

OBLON, SPIVAK, McCLELLAND,
MAIER & NEUSTADT, P.C.

Eckhard H. Kuesters

Registration No. 28,870

Customer Number

22850

(703) 413-3000 (phone)
(703) 413-2220 (fax)

1940 DUKE STREET ALEXANDRIA, VIRGINIA 22314 U.S.A.
TELEPHONE: 703-413-3000 FACSIMILE: 703-413-2220 WWW.OBLON.COM

BEST AVAILABLE COPY

DOCKET NO: 0039-7646-2RD



IN THE UNITED STATES PATENT & TRADEMARK OFFICE

IN RE APPLICATION OF :

TATSUNORI KANAI, ET AL. :

EXAMINER: JEAN, F.

SERIAL NO: 09/532,535 :

FILED: MARCH 22, 2000 :

GROUP ART UNIT: 2151

FOR: SCHEME FOR SYSTEMATICALLY :
REGISTERING META-DATA WITH
RESPECT TO VARIOUS TYPES OF
DATA

PETITION UNDER 37 C.F.R. § 1.181(A)(3)
TO INVOKE THE SUPERVISORY AUTHORITY OF THE COMMISSIONER

COMMISSIONER FOR PATENTS
ALEXANDRIA, VIRGINIA 22313

SIR:

Applicant herein petitions the Commissioner to invoke his supervisory authority to require the examiner to consider the prior art cited in the Information Disclosure Statements filed respectively on March 22, 2000, and April 30, 2002.

An Information Disclosure Statement in conformity with the requirements of 37 C.F.R. § 1.97 and § 1.98 was filed with the application on March 22, 2000 and, separately, on April 30, 2002. A copy of the Information Disclosure Statement, the List of References Cited by Applicants (i.e., PTO- Form 1449), the cited prior art references, and a date-stamped filing receipt are attached. The above-referenced application has now been allowed, however, the Information Disclosure Statements were never acknowledged or made of record by the examiner.

Application No. 09/532,535
Petition Under 37 C.F.R. § 1.181

Thus, this Petition is being filed in order to require the Examiner to consider the references listed on the Information Disclosure Statement.

Although Applicants do not believe that any fee is required for the present petition, any required fee should be charged the undersigned attorneys account no. 15-0030.

Respectfully submitted,

OBLON, SPIVAK, McCLELLAND,
MAIER & NEUSTADT, P.C.



Eckhard H. Kuesters
Attorney of Record
Registration No. 28,870

Customer Number

22850

Tel: (703) 413-3000
Fax: (703) 413 -2220
(OSMMN 06/04)

I:\ATTY\SAE\PROSECUTION\0039-7646\0039-7646-1.181 PETITION.DOC



OSM. /&N File No. 0039-7646-2RD

Dept.: PP

Serial No. New Application

By: MJS/slc

In the matter of the Application of: Tatsunori KANAI, et al.

For: SCHEME FOR SYSTEMATICALLY REGISTERING META-DATA WITH RESPECT TO VARIOUS TYPES OF DATA

The following has been received in the U.S. Patent Office on the date stamped hereon:

- ☒ 38 pp. Specification & 20 Claims/Drawings 11 Sheets
- ☒ Combined Declaration, Petition & Power of Attorney 5 pages
- ☐ List of Inventor Names and Addresses
- ☒ Utility Patent Application ☐ CPA
- ☒ Notice of Priority ☒ Priority Doc (1)
- ☒ Check for \$846.00 ☒ Dep. Acct. Order Form
- ☒ Fee Transmittal Form
- ☐ Assignment/PTO 1595 pages:
- ☐ Letter to Official Draftsman
- ☐ Letter Requesting Approval of Drawing Changes
- ☐ Drawings sheets ☐ Formal
- ☐ Letter
- ☐ Amendment
- ☒ Information Disclosure Statement ☒ PTO-1449
- ☒ Cited References (7)
- ☐ Search Report
- ☒ Statement of Relevancy
- ☐ IDS/Related/List of Related Cases
- ☐ Restriction Response ☐ Election Response
- ☐ Rule 132 Declaration
- ☐ Petition for Extension of Time
- ☐ Notice of Appeal
- ☐ Brief
- ☐ Issue Fee Transmittal
- ☒ White Advance Serial Number Card
- ☐
- ☐



Due Date: 03/23/00

BEST AVAILABLE COPY

COPY

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

IN RE APPLICATION OF:

Tatsunori KANAI, et al.

SERIAL NO: New Application

GAU:

FILED: Herewith

EXAMINER:

FOR: SCHEME FOR SYSTEMATICALLY REGISTERING META-DATA WITH RESPECT TO VARIOUS TYPES OF DATA

INFORMATION DISCLOSURE STATEMENT UNDER 37 CFR 1.97

ASSISTANT COMMISSIONER FOR PATENTS
WASHINGTON, D.C. 20231

SIR:

Applicant(s) wish to disclose the following information.

REFERENCES

- ☒ The applicant(s) wish to make of record the references listed on the attached form PTO-1449. Copies of the listed references are attached, where required, as are either statements of relevancy or any readily available English translations of pertinent portions of any non-English language references.
- ☐ A check is attached in the amount required under 37 CFR §1.17(p).

RELATED CASES

- ☐ Attached is a list of applicant's pending application(s) or issued patent(s) which may be related to the present application. A copy of the patent(s) is attached along with PTO 1449.
- ☐ A check is attached in the amount required under 37 CFR §1.17(p).

CERTIFICATION

- ☐ Each item of information contained in this information disclosure statement was cited in a communication from a foreign patent office in a counterpart foreign application not more than three months prior to the filing of this statement.
- ☐ No item of information contained in this information disclosure statement was cited in a communication from a foreign patent office in a counterpart foreign application or, to the knowledge of the undersigned, having made reasonable inquiry, was known to any individual designated in 37 CFR §1.56(c) more than three months prior to the filing of this statement.

PETITION

- ☐ Applicant(s) hereby request consideration of the attached information. A check is attached in the amount of the Petition fee required under 37 CFR §1.17(i)(1).

DEPOSIT ACCOUNT

- ☒ Please charge any additional fees for the papers being filed herewith and for which no check is enclosed herewith, or credit any overpayment to deposit account number 15-0030. A duplicate copy of this sheet is enclosed.

Respectfully submitted,

OBLON, SPIVAK, McCLELLAND,
KAMLER & NEUSTADT, P.C.

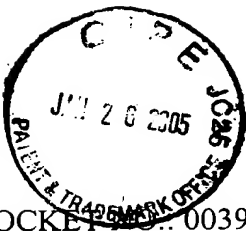
COPY

Marvin J. Spivak
Registration No. 24,913

Fourth Floor
1755 Jefferson Davis Highway
Arlington, Virginia 22202
Tel. (703) 413-3000
Fax. (703) 413-2220
(OSMMN 10/98)

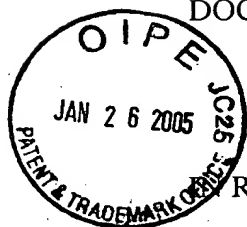
Form PTO 1449 (Modified)		U.S. DEPARTMENT OF COMMERCE PATENT AND TRADEMARK OFFICE		ATTY DOCKET NO. 0039-7646-2RD		SERIAL NO. New Application	
LIST OF REFERENCES CITED BY APPLICANT				APPLICANT Tatsunori KANAI, et al.			
				FILING DATE Herewith		GROUP	
OTHER REFERENCES (Including Author, Title, Date, Pertinent Pages, etc.)							
	AA	Y. Y. GOLAND, et al., "PRACTICAL FILE SYSTEM DESIGN WITH THE BE FILE SYSTEM", 1999, pgs. 65-97					
	AB	Y. Y. GOLAND, et al., "HTTP EXTENTIONS FOR DISTRIBUTED AUTHORIZING-WEBDAV", February 1999, pgs. 1-71					
	AC	"RECORDING-HELICAL-SCAN DIGITAL VIDEO CASSETTE RECORDING SYSTEM USING 6,35 MM MAGNETIC TAPE FOR CONSUMER USE (525-60, 625-50, 1125-60, 1250-50 SYSTEMS) - PART 4: PACK HEADER TABLE AND CONTENTS", International Electrotechnical Commission, 1998, pgs.1-7, 116-147					
	AD	Saveen REDDY, et al., "DAV SEARCHING AND LOCATING", DAV Searching and Locating Protocol, June 3, 1999, pgs. 1-26					
	AE	"DIGITAL STILL CAMERA IMAGE FILE FORMAT STANDARD", Japan Electronic Industry Development Association Standard, June 1998, pgs. 17-69					
	AF	T. BERNERS-LEE, et al., "Hypertext Transfer Protocol-HTTP/1.0", Network Working Group, May 1996, pgs. 1-60					
	AG	R. FIELDING, et al., "Hypertext Transfer Protocol-HTTP/1.1", Network Working Group, January 1997, pgs. 1-162					
	AH						
	AI						
	AJ						
	AK						
	AL						
	AM						
	AN						
	AO						
	AP						
	AQ						
Examiner						Date Considered	
*Examiner: Initial if reference is considered, whether or not citation is in conformance with MPEP 609; Draw line through citation if not in conformance and not considered. Include copy of this form with next communication to applicant.							

COPY



DOCKET NO.: 0039-7646-2RD

page 1 of 1



IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

RE APPLICATION OF: Tatsunori KANAI, et al.

SERIAL NO.: New Application

FILED: Herewith

FOR: SCHEME FOR SYSTEMATICALLY REGISTERING META-DATA WITH
RESPECT TO VARIOUS TYPES OF DATA

STATEMENT OF RELEVANCY

Reference AA on Form PTO-1449:

This reference relates to a file system where each file is managed with meta data (attributes and values). But, it doesn't show automatic mechanism to put data type specific meta data.

Reference AB on Form PTO-1449:

This discloses standard interface to put/get meta data on a resource via HTTP protocol.

COPY

BEST AVAILABLE COPY



BEST AVAILABLE COPY

*E
Zm*

Dept.: PP

OSMM&N File No. 0039-7646-2RD

By: MJS/atp/rm

Serial No. 09/532,535

In the matter of the Application of: Tatsunori KANAI, et al.

For: SCHEME FOR SYSTEMATICALLY REGISTERING META-DATA
WITH RESPECT TO VARIOUS TYPES OF DATA

The following has been received in the U.S. Patent Office on the date stamped hereon:

- ☐ pp. Specification & Claims/Drawings Sheets
- ☐ Combined Declaration, Petition & Power of Attorney pages
- ☐ List of Inventor Names and Addresses
- ☐ Utility Patent Application ☐ CPA
- ☐ Notice of Priority ☐ Priority Doc
- ☐ Check for ☒ Dep. Acct. Order Form
- ☐ Fee Transmittal Form
- ☐ Assignment/PTO-1595 pages:
- ☐ Letter to Official Draftsman
- ☐ Letter Requesting Approval of Drawing Changes
- ☐ Drawings sheets ☐ Formal
- ☐ Letter
- ☐ Amendment
- ☒ Information Disclosure Statement ☒ PTO-1449
- ☒ Cited References (12)
- ☒ EUROPEAN Search Report
- ☐ Statement of Relevancy
- ☐ English Abstracts, Concise Explanation, English Translation, Partial English Translation ()
- ☐ IDS/Related/List of Related Cases ☐ Cited Pending Applications ()
- ☐ Restriction Response ☐ Election Response
- ☐ Rule 132 Declaration
- ☐ Petition for Extension of Time
- ☐ Notice of Appeal



Due date: 06/18/02

COPY

Docket No. 0039-7646-2RD/atp

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

IN RE APPLICATION OF: Tatsunori KANAI, et al.

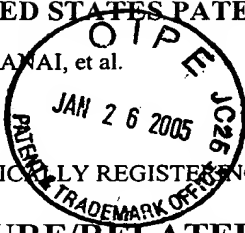
SERIAL NO: 09/532,535

GAU: 2755

FILED: March 22, 2000

EXAMINER:

FOR: SCHEME FOR SYSTEMATICALLY REGISTERING META-DATA WITH RESPECT TO VARIOUS TYPES OF DATA



INFORMATION DISCLOSURE/RELATED CASE STATEMENT UNDER 37 CFR 1.97

ASSISTANT COMMISSIONER FOR PATENTS
WASHINGTON, D.C. 20231

SIR:

Applicant(s) wish to disclose the following information.

REFERENCES

- ☒ The applicant(s) wish to make of record the references cited in the attached European Search Report listed on the attached form PTO-1449. Copies of the listed references are attached, where required, as are either statements of relevancy or any readily available English translations of pertinent portions of any non-English language references.
- ☐ A check is attached in the amount required under 37 CFR §1.17(p).

RELATED CASES

- ☐ Attached is a list of applicant's pending application(s) or issued patent(s) which may be related to the present application. A copy of the patent(s), together with a copy of the claims and drawings of the pending application(s) is attached along with PTO 1449.
- ☐ A check is attached in the amount required under 37 CFR §1.17(p).

CERTIFICATION

- ☒ Each item of information contained in this information disclosure statement was first cited in any communication from a foreign patent office in a counterpart foreign application not more than three months prior to the filing of this statement.
- ☐ No item of information contained in this information disclosure statement was cited in a communication from a foreign patent office in a counterpart foreign application or, to the knowledge of the undersigned, having made reasonable inquiry, was known to any individual designated in 37 CFR §1.56(c) more than three months prior to the filing of this statement.

DEPOSIT ACCOUNT

- ☒ Please charge any additional fees for the papers being filed herewith and for which no check is enclosed herewith, or credit any overpayment to deposit account number 15-0030. A duplicate copy of this sheet is enclosed.

Respectfully submitted,

OBLON, SPIVAK, McCLELLAND,
MAIER & NEUSTADT, P.C.

COPY

Marvin J. Spivak

Registration No.

24,913



22850

Tel. (703) 413-3000
Fax. (703) 413-2220
(OSMMN 10/98)

Form PTO 1449 (Modified)		U.S. DEPARTMENT OF COMMERCE PATENT AND TRADEMARK OFFICE		ATTY DOCKET NO. 0039-7646-2RD		SERIAL NO. 09/532,535	
LIST OF REFERENCES CITED BY APPLICANT				APPLICANT Tatsunori KANAI, et al.			
				FILING DATE March 22, 2000		GROUP 2755	
U.S. PATENT DOCUMENTS							
EXAMINER INITIAL		DOCUMENT NUMBER	DATE	NAME	CLASS	SUB CLASS	FILING DATE IF APPROPRIATE
	AA	5,715,397	02/03/98	S. S. OGAWA, et al.			
	AB	5,629,846	05/13/97	A. W. CRAPO			
	AC	5,627,997	05/06/97	M. E. PEARSON, et al.			
	AD	5,557,780	09/17/96	A. T. EDWARDS, et al.			
	AE	5,835,712	11/10/98	R. B. DuFRESNE			
	AF	5,721,912	02/24/98	F. M. STEPCZYK, et al.			
	AG						
	AH						
	AI						
	AJ						
FOREIGN PATENT DOCUMENTS							
		DOCUMENT NUMBER	DATE	COUNTRY	TRANSLATION YES NO		
	AK	53031/98	08/27/98	AUSTRALIA			
	AL	WO 98/03928	01/29/98	WIPO			
	AM						
	AN						
	AO						
	AP						
	AQ						
OTHER REFERENCES (Including Author, Title, Date, Pertinent Pages, etc.)							
	AR	R. A. NADO, et al., SIGMOD Record, vol. 26, no. 4, pages 32 -38, XP-002193377, "EXTRACTING ENTITY PROFILES FROM SEMISTRUCTURED INFORMATION SPACES", December 1997					
	AS	B. ADELBERG, ACM Proceedings of SIGMOD. International Conference on Management of Data, vol. 27, no. 2, pages 1 - 25, XP-002949327, "NoDoSE - A TOOL FOR SEMI-AUTOMATICALLY EXTRACTING STRUCTURED AND SEMISTRUCTURED DATA FROM TEXT DOCUMENTS", 1998					
	AT	N. ASHISH, et al., Proceedings of the Second IFCIS International Conference on Kiawah Island, pages 160 - 169, XP-010240791, "SEMI-AUTOMATIC WRAPPER GENERATION FOR INTERNET INFORMATION SOURCES", 1997					
	AU	D. FLORESCU, et al., SIGMOD Record, vol. 27, no. 3, pages 59-74, XP-002193378, "DATABASE TECHNIQUES FOR THE WORLD-WIDE WEB: A SURVEY", September 1998					
	AV	COPY					
	AW						
Examiner					Date Considered		
<small>*Examiner: Initial if reference is considered, whether or not citation is in conformance with MPEP 609; Draw line through citation if not in conformance and not considered. Include copy of this form with next communication to applicant.</small>							

J738-508-V5
BAZD

F

Practical File System Design

with the Be File System

Dominic Giampaolo

Be, Inc.

BEST AVAILABLE COPY



MORGAN KAUFMANN PUBLISHERS, INC.
San Francisco, California

Editor Tim Cox
 Director of Production and Manufacturing Yonle Overton
 Assistant Production Manager Julie Pabst
 Editorial Assistant Sarah Luger
 Cover Design Ross Caron Design
 Cover Image William Thompson/Photonica
 Copyeditor Ken DellaPenta
 Proofreader Jennifer McClain
 Text Design Side by Side Studios
 Illustration Cherie Plumlee
 Composition Ed Szyner, Babel Press
 Indexer Ty Koontz
 Printer Edwards Brothers

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances where Morgan Kaufmann Publishers, Inc. is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

Morgan Kaufmann Publishers, Inc.
 Editorial and Sales Office
 340 Pine Street, Sixth Floor
 San Francisco, CA 94104-3205
 USA
 Telephone 415/392-2665
 Facsimile 415/392-2665
 Email mktg@mkp.com
 WWW <http://www.mkp.com>
 Order toll free 800/745-7323

©1999 Morgan Kaufmann Publishers, Inc.
 All rights reserved
 Printed in the United States of America

03 02 01 00 99 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, recording, or otherwise—without the prior written permission of the publisher.

Library of Congress Cataloging-in-Publication Data is available for this book.
 ISBN 1-55860-497-9

Contents

Preface

Chapter 1 Introduction to the BeOS and BFS

- 1.1 History Leading Up to BFS
- 1.2 Design Goals
- 1.3 Design Constraints
- 1.4 Summary

Chapter 2 What Is a File System?

- 2.1 The Fundamentals
- 2.2 The Terminology
- 2.3 The Abstractions
- 2.4 Basic File System Operations
- 2.5 Extended File System Operations
- 2.6 Summary

Chapter 3 Other File Systems

- 3.1 BSD FFS
- 3.2 Linux ext2
- 3.3 Macintosh HFS
- 3.4 Irix XFS
- 3.5 Windows NT's NTFS
- 3.6 Summary

Chapter 4 The Data Structures of BFS

- 4.1 What Is a Disk?
- 4.2 How to Manage Disk Blocks
- 4.3 Allocation Groups
- 4.4 Block Runs

ix

1

1

4

5

5

7

7

8

9

20

28

31

33

33

36

37

38

40

44

45

45

46

46

47

v

did it mesh well with the B+tree routines. The B+tree routines also have a notion of page size (although it is completely independent of the rest of the file system). The B+tree routines have a restriction that the maximum size of a stored item must be less than half the B+tree page size. Since BFS allows 255-character file names, the B+tree page size also had to be at least 1024 bytes. Pushing the minimum file system block size to 1024 bytes ensures that i-nodes have sufficient space to store a reasonable number of attributes and that the B+tree pages correspond nicely to file system blocks so that allocation and I/O done on behalf of the B+trees does not need any additional messaging.

You may ask, If 1024 bytes is a good file system block size, why not jump to 2048 bytes? I did experiment with 2048-byte blocks and 4096-byte blocks. The additional space available for attributes was not often used (an email message uses on average about 500 bytes to store nine attributes). B-trees also presented a problem as their size grew significantly with a 2048-byte page size: a balanced B+tree tends to be half full, so on average each page of a B+tree would have only 1024 bytes of useful data. Some quick experiments showed that directory and index sizes grew much larger than desirable with a 2048-byte page size. The conclusion was that although larger block sizes have desirable properties for very large files, the added cost for normal files was not worthwhile.

The allocation group concept also underwent considerable revision. Originally the intent was that each allocation group would allow operations to take place in parallel in the file system; that is, each allocation group would appear as a mini file system. Although still very attractive (and it turns out quite similar to the way the Linux ext2 file system works), the reality was that journaling forced serialization of all file system modifications. It might have been possible to have multiple logs, one per allocation group; however, that idea was not pursued because of a lack of time.

The original intent of the allocation group concept was for very large allocation groups (about eight per gigabyte). However, this proved unworkable for a number of reasons: first and foremost, the `block.run` data structure only had a 16-bit starting block number, and further, such a small number of allocation groups didn't carve the disk into enough chunks. Instead the number of allocation groups is a factor of the number of bitmap blocks required to map 65,536 blocks. By sizing the allocation groups this way, we allow maximum use of the `block.run` data structure.

It is clear that many factors influence design decisions about the size, layout, and organization of file system data structures. Although decisions may be based on intuition, it is important to verify that those decisions make sense by looking at the performance of several alternatives.

This introduction to the raw data structures that make up BFS lays the foundation for understanding the higher-level concepts that go into making a complete file system.

Attributes, Indexing, and Queries

This chapter is about three closely related topics: attributes and indexing of attributes. In combination these three features add considerable power to a file system and endow the file system with many of the features normally associated with a database. This chapter aims to show why attributes, indexing, and queries are an important feature of a modern file system. We will discuss the high-level issues as well as the details of the BFS implementation.

5.1 Attributes

What are attributes? In general an attribute is a name (usually a short descriptive string) and a value such as a number, string, or even raw binary data. For example, an attribute could have a name such as Age and a value of 27 or a name of Keywords and a value of Computers File System Journaling. An attribute is information about an entity. In the case of a file system, an attribute is additional information about a file that is not stored in the file itself. The ability to store information about a file with the file but not in it is very important because often modifying the contents of a file to store the information is not feasible—or even possible.

There are many examples of data that programs can store in attributes:

- Icon position and information for a window system
- The URL of the source of a downloaded Web document
- The type of a file
- The last backup date of a file
- The "To," "From," and "Subject" lines of an email message
- Keywords in a document

- Access control lists for a security system
- Style information for a styled text editor (fonts, sizes, etc.)
- Gamma correction, color depth, and dimensions of an image
- A comment about a file
- Contact database information (address, phone/fax numbers, email address, URL)

These are examples of information about an object, but they are not necessarily information we would—or even could—store in the object itself. These examples just begin to touch upon the sorts of information we might store in an attribute. The ability to attach arbitrary name/value pairs to a file opens up many interesting possibilities.

Examples of the Use of Attributes

Consider the need to manage information about people. An email program needs an email address for a person, a contact manager needs a phone number, a fax program needs a fax number, and a mail-merge for a word processor needs a physical address. Each of these programs has specific needs, and generally each program would have its own private copy of the information it needs about a person, although much information winds up duplicated in each application. If some piece of information about a person should change, it requires updating several different programs—not an ideal situation.

Instead, using attributes, the file system can represent the person as a file. The name of the file would be the name of the person or perhaps a more unique identifier. The attributes of this “person file” can maintain the information about the person: the email address, phone number, fax number, URL, and so on. Then each of the programs mentioned above simply accesses the attributes that it needs. All of the programs go to the same place for the information. Further, programs that need to store different pieces of information can add and modify other attributes without disturbing existing programs.

The power of attributes in this example is that many programs can share information easily. Because access to attributes is uniform, the applications must agree on only the names of attributes. This facilitates programs working together, eliminates wasteful duplication of data, and frees programs from all having to implement their own minidatabase. Another benefit is that new applications that require previously unknown attributes can add the new attributes without disrupting other programs that use the older attributes.

In this example, other benefits also accrue by storing the information as attributes. From the user's standpoint a single interface exists to information about people. They can expect that if they select a person in an email program, the email program will use the person's email attribute and allow the user to send them email. Likewise if the user drags and drops the icon of a

“person file” onto a fax program, it is natural to expect that the fax program will know that you want to send a fax to that person. In this example, attributes provide an easy way to centralize storage of information about people and to do it in a way that facilitates sharing it between applications.

Other less sophisticated examples abound. A Web browser could store the URL of the source of a downloaded file to allow users to later ask, “Go back to the site where this file came from.” An image-scanning program could store color correction information about a scan as an attribute of the file. A text editor that uses fonts and styles could store the style information about the text as an attribute, leaving the original text as plain ASCII (this would enable editing source code with multiple fonts, styles, colors, etc.). A text editor could synthesize the primary keywords contained in a document and store those as attributes of the document so that later files could be searched for a certain type of content.

These examples all illustrate ways to use attributes. Attributes provide a mechanism for programs to store data about a file in a way that makes it easy to later retrieve the information and to share it with other applications.

Attribute API

Many operations on attributes are possible, but the file system interface in the BeOS keeps the list short. A program can perform the following operations on file attributes:

- Write attribute
- Read attribute
- Open attribute directory
- Read attribute directory
- Rewind attribute directory
- Close attribute directory
- Stat attribute
- Remove attribute
- Rename attribute

Not surprisingly, these operations bear close resemblance to the corresponding operations for files, and their behavior is virtually identical. To access the attributes of a file, a program must first open the file and use that file descriptor as a handle to access the attributes. The attributes of a file do not have individual file descriptors. The attribute directory of a file is similar to a regular directory. Programs can open it and iterate through it to enumerate all the attributes of a file.

Notably absent from the list are operations to open and close attributes as we would with a regular file. Because attributes do not use separate file descriptors for access, open and close operations are superfluous. The user-level API calls to read and write data from attributes have the following prototypes:

```

ssize_t fs_read_attr(int fd, const char *attribute, uint32 type,
                    off_t pos, void *buf, size_t count);

ssize_t fs_write_attr(int fd, const char *attribute, uint32 type,
                     off_t pos, const void *buf, size_t count);

```

Each call encapsulates all the state necessary to perform the I/O. The file descriptor indicates which file to operate on; the attribute name indicates which attribute to do the I/O to; the type indicates the type of data being written, and the position specifies the offset into the attribute to do the I/O at. The semantics of the attribute read/write operations are identical to file read/write operations. The write operation has the additional semantics that if the attribute name does not exist, it will create it implicitly. Writing to an attribute that exists will overwrite the attribute (unless the position is nonzero, and then it will extend the attribute if it already exists).

The functions to list the attributes of a file correspond very closely with the standard POSIX functions to list the contents of a directory. The open attribute directory operation initiates access to the list of attributes belonging to a file. The open attribute directory operation returns a file descriptor because the state associated with reading a directory cannot be maintained in user space. The read attribute directory operation returns the next successive entry until there are no more. The rewind operation resets the position in the directory stream to the beginning of the directory. Of course, the close operation simply closes the file descriptor and frees the associated state.

The remaining operations (stat, remove, and rename) are typical house-keeping operations and have no subtleties. The stat operation, given a file descriptor and attribute name, returns information about the size and type of the attribute. The remove operation deletes the named attribute from the list of attributes associated with a file. The rename operation is not currently implemented in BFS.

Attribute Details

As defined previously, an attribute is a string name and some arbitrary chunk of data. In the BeOS, attributes also declare the type of the data stored with the name. The type of the data is either an integral type (string, integer, or floating-point number) or it is simply raw data of arbitrary size. The type field is only strictly necessary to support indexing.

In deciding what data structure to use to store an attribute, our first temptation might be to define a new data structure. But if we resist that temptation and look closer at what an attribute must store, we find that the description is strikingly similar to that of a file. At the most basic level an attribute is a named entity that must store an arbitrary amount of data. Although it is true that most attributes are likely to be small, storing large amounts of data

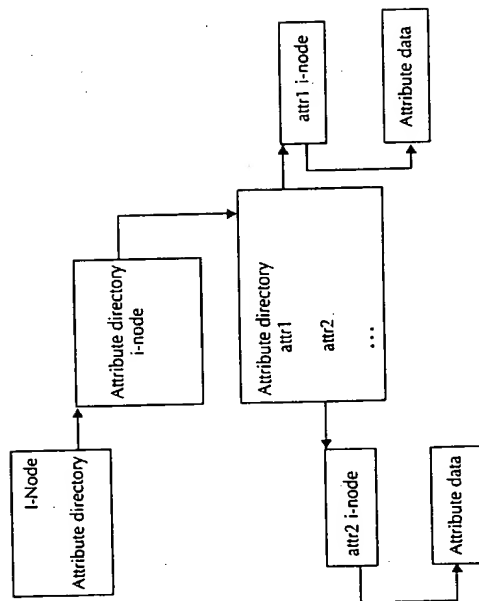


Figure 5-1 Relationship between an i-node and its attributes.

in an attribute is quite useful and needs full support. With this in mind it makes good sense to reuse the data structure that underlies files—an i-node. An i-node represents a stream of data on disk and thus can store an arbitrary amount of information. By storing the contents of an attribute in the data stream of an i-node, the file system does not have to manage a separate set of data structures specific to attributes.

The list of attributes associated with a file also needs a data structure and place for storage. Taking our cue from what we observed about the similarity of attributes to files, it is natural to store the list of attributes as a directory. A directory has exactly the properties needed for the task: it maps names to i-nodes. The final glue necessary to bind together all the structures is a reference from the file i-node to the attribute directory i-node. Figure 5-1 diagrams the relationships between these structures. Then it is possible to traverse from a file i-node to the directory that lists all the attributes. From the directory entries it is possible to find the i-node of each of the attributes, and having access to the attribute i-node gives us access to the contents of the attribute.

This implementation is the simplest to understand and implement. The only drawback to this approach is that, although it is elegant in theory, in practice its performance will be abysmal. Performance will suffer because each attribute requires several disk operations to locate and load. The initial design of BFS used this approach. When it was first presented to other engineers, it was quickly shot down (and rightly so) because of the levels of indirection necessary to reach an attribute.

This performance bottleneck is an issue because in the BeOS the window system stores icon positions for files as attributes of the file. Thus, with this design, when displaying all the files in a directory, each file would need at least one disk access to get the file i-node, one access to load the attribute directory i-node, another directory access to look up the attribute name, another access to load the attribute i-node, and finally yet another disk access to load the data of the attribute. Given that current disk drives have access times on the order of milliseconds (and sometimes tens of milliseconds) while CPU speeds reach into the sub-5-nanosecond range, it is clear that forcing the CPU to wait for five disk accesses to display a single file would devastate performance.

We knew that a number of the attributes of a file would be small and that providing quick access to them would benefit many programs. In essence the problem was that at least some of the attributes of a file needed more efficient access. The solution came together as another design issue reared its head at roughly the same time. BFS needed to be able to store an arbitrary number of files on a volume, and it was not considered acceptable to reserve space on a volume for i-nodes up front. Reserving space for i-nodes at file system initialization time is the traditional approach to managing i-nodes, but this can lead to considerable wasted space on large drives with few files and invariably can become a limitation for file systems with lots of files and not enough i-nodes. BFS needed to only consume space for as many or as few files as were stored on the disk—no more, no less. This implied that i-nodes would likely be stored as individual disk blocks. Initially it seemed that storing each i-node in its own disk block would waste too much space because the size of the i-node structure is only 232 bytes. However, when this method of storing i-nodes is combined with the need to store several small attributes for quick access, the solution is clear. The spare space of an i-node block is suitable for storage of small attributes of the file. BFS terms this space at the end of an i-node block as the `small_data` area. Conceptually a BFS i-node looks like Figure 5-2.

Because not all attributes can fit in the `small_data` area of an i-node, BFS continues to use the attribute directory and i-nodes to store additional attributes. The cost of accessing nonresident attributes is indeed greater than attributes in the `small_data` area, but the trade-off is well worth it. The most common case is extremely efficient because one disk read will retrieve the i-node and a number of small attributes that are often the most needed.

The `small_data` area is purely an implementation detail of BFS and is completely transparent to programmers. In fact, it is not possible to request that an attribute be put in the `small_data` area. Exposing the details of this performance tweak would mar the otherwise clean attribute API.

small_data Area Detail

The data structure BFS uses to manage space in the `small_data` area is

Main i-node information Name, size, modification time, ...
<code>small_data</code> area
<code>attr1</code>
<code>attr2</code>
<code>attr3</code>
...

Figure 5-2 A high-level view of a BFS i-node and `small_data` area.

```
typedef struct small_data {
    uint32  type;
    uint16  name_size;
    uint16  data_size;
    char    name[1];
} small_data;
```

This data structure is optimized for size so that as many as possible could be packed into the i-node. The two size fields, `name_size` and `data_size`, are limited to 16-bit integers because we know the size of the i-node will never be more than 8K. The type field would also be 16 bits but we must preserve the exact type passed in from higher-level software.

The content of the name field is variable sized and begins in the last field of the `small_data` structure (the member name in the structure is just an easy way to refer to the beginning of the bytes that constitute the name rather than a fixed-size name of only one character). The data portion of the attribute is stored in the bytes following the name with no padding. A C macro that yields a pointer to the data portion of the `small_data` structure is

```
#define SD_DATA(sd) \
    (void *)(((char *)sd + sizeof(*sd) + (sd->name_size-sizeof(sd->name))))
```

In typical obfuscated C programming fashion, this macro uses pointer arithmetic to generate a pointer to the bytes following the variable-sized name field. Figure 5-3 shows how the `small_data` area is used.

All routines that manipulate the `small_data` structure expect a pointer to an i-node, which in BFS is not just the i-node structure itself but the entire disk block that the i-node resides in. The following routines exist to manipulate the `small_data` area of an i-node:

- Find a `small_data` structure with a given name
- Create a new `small_data` structure with a name, type, and data

i-node #, size, owner, permissions, ...			bfs_i-node structure
type name_size	data_size name data	type	small_data area
name_size	data_size name data	type name_size	
data_size	name data	type name_size data_size	
name data	type name_size data_size name data		
Free space			

Figure 5-3 A BFS i-node, including the small_data area.

- Update an existing small_data structure
- Get the data portion of a small_data structure
- Delete a small_data structure

Starting from the i-node address, the address of the first small_data structure is easily calculated by adding the size of the i-node structure to its address. The resulting pointer is the base of the small_data area. With the address of the first small_data structure in hand, the routines that operate on the small_data area all expect and maintain a tightly packed array of small_data structures. The free space is always the last item in the array and is managed as a small_data item with a type of zero, a zero-length name, and a data_size equal to the size of the remaining free space (not including the size of the structure itself).

Because BFS packs the small_data structures as tightly as possible, any given instance of the small_data structure is not likely to align itself on a "nice" memory boundary (i.e., "nice" boundaries are addresses that are multiples of four or eight). This can cause an alignment exception on certain RISC processors. Were the BeOS to be ported to an architecture such as MIPS, BFS would have to first copy the small_data structure to a properly aligned temporary variable and dereference it from there, complicating the code considerably. Because the CPUs that the BeOS runs on currently (PowerPC and Intel x86) do not have this limitation, the current BFS code ignores the problem despite the fact that it is nonportable.

The small_data area of an i-node works well for storing a series of tightly packed attributes. The implementation is not perfect though, and there are other techniques BFS could have used to reduce the size of the small_data structure even further. For example, a C union type could have been employed to eliminate the size field for fixed-size attributes such as integers or floating-point numbers. Or the attribute name could have been stored as a hashed value, instead of the explicit string, and the string looked up in a

```
if length of data being written is small
    find the attribute name in the small_data area
    if found
        delete it from small_data and from any indices
    else
        create the attribute name

        write new data
        if it fits in the small_data area
            delete it from the attribute directory if present
        else
            create the attribute in the attribute directory
            write the data to the attribute i-node
            delete name from the small_data area if it exists

    else
        create the attribute in the attribute directory
        write the data to the attribute i-node
        delete name from the small_data area if it exists
```

Listing 5-1 Pseudocode for the write attribute operation of BFS.

hash table. Although these techniques would have saved some space, they would have complicated the code further and made it even more difficult to debug. As seemingly simple as it is, the handling of small_data attributes took several iterations to get correct.

The Big Picture: small_data Attributes and More

The previous descriptions provide ample detail of the mechanics of using the small_data structure but do not provide much insight into how this connects with the general attribute mechanisms of BFS. As we discussed earlier, a file can have any number of attributes, each of which is a name/value pair of arbitrary size. Internally the file system must manage attributes that reside in the small_data area as well as those that live in the attribute directory.

Conceptually managing the two sets of attributes is straightforward. Each time a program requests an attribute operation, the file system checks if the attribute is in the small_data area. If not, it then looks in the attribute directory for the attribute. In practice, though, this adds considerable complexity to the code. For example, the write attribute operation uses the algorithm shown in Listing 5-1.

Subtleties such as deleting the attribute from the attribute directory after adding it to the small_data area are necessary in situations where rewriting an existing attribute causes the location of the attribute to change.

Manipulating attributes that live in the attribute directory of a file is eased because many of the operations can reuse the existing operations that work on files. Creating an attribute in the attribute directory uses the same underlying functions that create a file in a directory. Likewise, the operations that read, write, and remove attributes do so using the same routines as files. The glue code necessary for these operations has subtleties analogous to the operations on the `small` data area (attributes need to be deleted from the `small` data area if they exist when an attribute is written to the attribute directory, and so on).

File system reentrancy is another issue that adds some complexity to the situation. Because the file system uses the same operations for access to the attribute directory and attributes, we must be careful that the same resources are not ever locked a second time (which would cause a deadlock). Fortunately, deadlock problems such as this are quite catastrophic if encountered, making it easy to detect when they happen (the file system locks up) and to correct (it is easy to examine the state of the offending code and to backtrack from there to a solution).

Attribute Summary

The basic concept of an attribute is a name and some chunk of data associated with that name. An attribute can be something simple:

Keywords - bass, guitar, drums

or it can be a much more complex piece of associated data. The data associated with an attribute is free-form and can store anything. In a file system, attributes are usually attached to files and store information about the contents of the file.

Implementing attributes is not difficult, although the straightforward implementation will suffer in performance. To speed up access to attributes, BFS supports a fast-attribute area directly in the i-node of a file. The fast-attribute area significantly reduces the cost of accessing an attribute.

5.2 Indexing

To understand indexing it is useful to imagine the following scenario: Suppose you went to a library and wanted to find a book. At the library, instead of a meticulously organized card catalog, you found a huge pile of cards, each card complete with the information (attributes) about a particular book. If there was no order to the pile of cards, it would be quite tedious to find the book you wanted. Since librarians prefer order to chaos, they keep three indices of information about books. Each catalog is organized alphabetically, one by book title, one by author name, and one by subject area. This makes

it rather simple to locate a particular book by searching the author, title, or subject index cards.

Indexing in a file system is quite similar to the card catalog in a library. Each file in a file system can be thought of as equivalent to a book in a library. If the file system does not index the information about a file, then finding a particular file can result in having to iterate over all files to find the one that matches. When there are many files, such an exhaustive search is slow. Indexing items such as the name of a file, its size, and the time it was last modified can significantly reduce the amount of time it takes to find a file.

In a file system, an index is simply a list of files ordered on some criteria. With the presence of additional attributes that a file may have, it is natural to allow indexing of other attributes besides those inherent to the file. Thus a file system could index the Phone Number attribute of a person, the From field of email addresses, or the Keywords of a document. Indexing additional attributes opens up considerable flexibility in the ways in which users can locate information in a file system.

If a file system indexes attributes about a file, a user can ask for sophisticated queries such as "find all email from Bob Lewis received in the last week." The file system can search its indices and produce the list of files that match the criteria. Although it is true that an email program could do the same, doing the indexing in the file system with a general-purpose mechanism allows all applications to have built-in database functionality without requiring them to each implement their own database.

A file system that supports indexing suddenly takes on many characteristics of a traditional database, and the distinction between the two blurs. Although a file system that supports attributes and indexing is quite similar to a database, the two are not the same because their goals push the two in subtly different directions. For example, a database trades some flexibility (a database usually has fixed-size entries, it is difficult to extend a record after the database is created, etc.) for features (greater speed and ability to deal with larger numbers of entries, richer query interface). A file system offers more generality at the expense of overhead: storing millions of 128-byte records as files in a file system would have considerable overhead. So although on the surface a file system with indices and a database share much functionality, the different design goals of each keep them distinct.

By simplifying many details, the above examples give a flavor for what is possible with indices. The following sections discuss the meatier issues involved.

What Is an Index?

The first question we need to answer is, What is an index? An index is a mechanism that allows efficient lookups of input values. Using our card catalog example, if we look in the author index for "Donald Knuth," we will

find references to books written by Donald Knuth, and the references will allow us to locate the physical copy of the book. It is efficient to look up the value "Knuth" because the catalog is in alphabetical order. We can jump directly to the section of cards for authors whose name begins with "K" and from there jump to those whose name begins with "Kn" and so on.

In computer terms, an index is a data structure that stores key/value pairs and allows efficient lookups of keys. The key is a string, integer, floating-point number, or other data item that can be compared. The value stored with a key is usually just a reference to the rest of the data associated with the key. For a file system the value associated with a key is the i-node number of the file associated with the key.

The keys of an index must always have a consistent order. That is, if the index compares key A against key B, they must always have the same relation—either A is less than B, greater than B, or equal to B. Unless the value of A or B changes, their relation cannot change. With integral computer types such as strings and integers, this is not a problem. Comparing more complex structures can make the situation less clear.

Many textbooks expound on different methods of managing sorted lists of data. Usually each approach to keeping a sorted list of data has some advantages and some disadvantages. For a file system there are several requirements that an indexing data structure must meet:

- It must be an on-disk structure.
- It must have a reasonable memory footprint.
- It must have efficient lookups.
- It must support duplicate entries.

First, any indexing method used by a file system must inherently be an on-disk data structure. Most common indexing methods only work in memory, making them inappropriate for a file system. File system indices must exist on permanent storage so that they will survive reboots and crashes. Further, because a file system is merely a supporting piece of an entire OS and not the focal point, using indices cannot impose undue requirements on the rest of the system. Consequently, the entire index cannot be kept in memory nor can a significant chunk of it be loaded each time the file system accesses an index. There may be many indices on a file system, and a file system needs to be able to have any number of them loaded at once and be able to switch between them as needed without an expensive performance hit each time it accesses a new index. These constraints eliminate from consideration a number of indexing techniques commonly used in the commercial database world.

The primary requirement of an index is that it can efficiently look up keys. The efficiency of the lookup operation can have a dramatic effect on the overall performance of the file system because every access to a file name must

perform a lookup. Thus it is clear that lookups must be the most efficient operation on an index.

The final requirement, and perhaps the most difficult, is the need to support duplicate entries in an index. At first glance, support for duplicate entries may seem unnecessary, but it is not. For example, duplicate entries are indispensable if a file system indexes file names. There will be many duplicate names because it is possible for files to have the same name if they live in different directories. Depending on the usage of the file system, the number of duplicates may range from only a few per index to many tens of thousands per index. Performance can suffer greatly if this issue is not dealt with well.

Data Structure Choices

Although many indexing data structures exist, there are only a few that a file system can consider. By far the most popular data structure for storing an on-disk index is the B-tree or any of its variants (B*-tree, B+tree, etc.). Hash tables are another technique that can be extended to on-disk data structures. Each of these data structures has advantages and disadvantages. We'll briefly discuss each of the data structures and their features.

B-trees

A B-tree is a treelike data structure that organizes data into a collection of nodes. As with real trees, B-trees begin at a root, the starting node. Links from the root node refer to other nodes, which, in turn, have links to other nodes, until the links reach a leaf node. A leaf node is a B-tree node that has no links to other nodes.

Each B-tree node stores some number of key/value pairs (the number of key/value pairs depends on the size of the node). Alongside each key/value pair is a link pointer to another node. The keys in a B-tree node are kept in order, and the link associated with a key/value pair points to a node whose keys are all less than the current key.

Figure 5-4 shows an example of a B-tree. Here we can see that the link associated with the word *cat* points to nodes that only contain values lexicographically less than the word *cat*. Likewise, the link associated with the word *indigo* refers to a node that contains a value less than *indigo* but greater than *deluxe*. The bottom row of nodes (*able*, *ball*, etc.) are all leaf nodes, because they have no links.

One important property of B-trees is that they maintain a relative ordering between nodes. That is, all the nodes referred to by the link from *man* in the root node will have entries greater than *cat* and less than *man*. The B-tree search routine takes advantage of this property to reduce the amount of work needed to find a particular node.

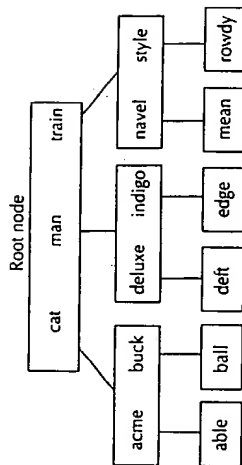


Figure 5-4 An example B-tree.

Knowing that B-tree nodes are sorted and the links for each entry point to nodes with keys less than the current key, we can perform a search of the B-tree. Normally searching each node uses a binary search, but we will illustrate using a sequential search to simplify the discussion. If we wanted to find the word *deft* we would start at the root node and search through its keys for the word *deft*. The first key, *cat*, is less than *deft*, so we continue. The word *deft* is less than *man*, so we know it is not in this node. The word *man* has a link though, so we follow the link to the next node. At the second-level node (*deluxe indigo*) we compare *deft* against *deluxe*. Again, *deft* is less than *deluxe*, so we follow the associated link. The final node we reach contains the word *deft*, and our search is successful. Had we searched for the word *depend*, we would have followed the link from *deluxe* and discovered that our key was greater than *deft*, and thus we would have stopped the search because we reached a leaf node and our key was greater than all the keys in the node.

The important part to observe about the search algorithm is how few nodes we needed to look at to do the search (3 out of 10 nodes). When there are many thousands of nodes, the savings is enormous. When a B-tree is well balanced, as in the above example, the time it takes to search a tree of N keys is proportional to $\log_2(N)$. The base of the logarithm, k , is the number of keys per node. This is a very good search time when there are many keys and is the primary reason that B-trees are popular as an indexing technique.

The key to the performance of B-trees is that they maintain a reasonable balance. An important property of B-trees is that no one branch of the tree is significantly taller than any other branch. Maintaining this property is a requirement of the insertion and deletion operations, which makes their implementation much more complex than the search operation.

Insertion into a B-tree first locates the desired insertion position (by doing a search operation), and then it attempts to insert the key. If inserting the key would cause the node to become overfull (each node has a fixed maximum size), then the node is split into two nodes, each getting half of the keys. Splitting a node requires modifications to the parent nodes of the node that

is split. The parent nodes of a split node need to change their pointers to the child node because there are now two. This change may propagate all the way back up to the root node, perhaps even changing the root node (and thus creating a new root).

Deletion from a B-tree operates in much the same way as insertion. Instead of splitting a node, however, deletion may cause pairs of nodes to coalesce into a single node. Merging adjacent nodes requires modification of parent nodes and may cause a similar rebalancing act as happens with insertions.

These descriptions of the insertion and deletion algorithms are not meant to be implementation guides but rather to give an idea of the process involved. If you are interested in this topic, you should refer to a file structures textbook for the specifics of implementing B-trees, such as Folk, Zoellick, and Riccardi's book.

Another benefit of B-trees is that their structure is inherently easy to store on disk. Each node in a B-tree is usually a fixed size, say, 1024 or 2048 bytes, a size that corresponds nicely to the disk block size of a file system. It is very easy to store a B-tree in a single file. The links between nodes in a B-tree are simply the offsets in the file of the other nodes. Thus if a node is located at position 15,360 in a file, storing a pointer to it is simply a matter of storing the value 15,360. Retrieving the node stored there requires seeking to that position in the file and reading the node.

As keys are added to a B-tree, all that is necessary to grow the tree is to increase the size of the file that contains the B-tree. Although it may seem that splitting nodes and rebalancing a tree may be a potentially expensive operation, it is not because there is no need to move significant chunks of data. Splitting a node into two involves allocating extra space at the end of the file, but the other affected nodes only need their pointers updated; no data must be rearranged to make room for the new node.

B-tree Variants

There are several variants of a standard B-tree, some of which have even better properties than traditional B-trees. The simplest modification, B⁺ trees, increases how full a node can be before it is split. By increasing the number of keys per node, we reduce the height of the tree and speed up searching.

The other more significant variant of a B-tree is a B+tree. A B+tree adds the restriction that all key/value pairs may only reside in leaf nodes. The interior nodes of a B+tree only contain index values to guide searches to the correct leaf nodes. The index values stored in the interior nodes are copies of the keys in the leaf nodes, but the index values are only used for searching, never for retrieval. With this extension, it is useful to link the leaf nodes together left to right (so, for example, in the B-tree defined above, the node *able* would contain a link to *ball*, etc.). By linking the leaf nodes together, it becomes easy to iterate sequentially over the contents of the B+tree. The other benefit is that interior nodes can have a different format than leaf nodes, making it

easy to pack as much data as possible into an interior node (which makes for a more efficient tree).

If the data being indexed is a string of text, another technique can be applied to compact the tree. In a prefix B+tree the interior nodes store only as much of the keys as necessary to traverse the tree and still arrive at the correct leaf node. This modification can reduce the amount of data that needs to be stored in the interior nodes. By reducing the amount of information stored in the interior nodes, the prefix B+tree stays shorter than if the compaction were not done.

Hashing

Hashing is another technique for storing data on disk. Hashing is a technique where the input keys are fed through a function that generates a hash value for the key. The same key value should always generate the same hash value. A hash function accepts a key and returns an integer value. The hash value of a key is used to index the hash table by taking the hash value modulo the size of the table to generate a valid index into the table. The items stored in the table are the key/value pairs just as with B-trees. The advantage of hashing is that the cost to look for an item is constant: the hash function is independent of the number of items in the hash table, and thus lookups are extremely efficient.

Except under special circumstances where all the input values are known ahead of time, the hash value for an input key is not always unique. Different keys may generate the same hash value. One method to deal with multiple keys colliding on the same hash value is to chain together in a linked list all the values that hash to the same table index (that is, each table entry stores a linked list of key/value pairs that map to that table entry). Another method is to rehash using a second hash function and to continue rehashing until a free spot is found. Chaining is the most common technique since it is the easiest to implement and has the most well-understood properties.

Another deficiency of hash tables is that hashing does not preserve the order of the keys. This makes an in-order traversal of the items in a hash table impossible.

One problem with hashing as an indexing method is that as the number of keys inserted into a table increases, so do the number of collisions. If a hash table is too small for the number of keys stored in it, then the number of collisions will be high and the cost of finding an entry will go up significantly (as the chain is just a linked list). A large hash table reduces the number of collisions but also increases the amount of wasted space (table entries with nothing in them). Although it is possible to change the size of a hash table, this is an expensive task because all the key/value pairs need to be rehashed. The expense of resizing a hash table makes it a very difficult choice for a general-purpose file system indexing method.

A variation on regular hashing, *extendible hashing*, divides a hash table into two parts. In extendible hashing there is a file that contains a directory of bucket pointers and a file of buckets (that contain the data). Extendible hashing uses the hash value of a key to index the directory of bucket pointers. Not all of the bits of the hash value are used initially. When a bucket overflows, the solution is to increase the number of bits of the hash value that are used as an index in the directory of bucket pointers. Increasing the size of the directory file is an expensive operation. Further, the use of two files complicates the use of extendible hashing in a file system.

Indexing in a file system should not waste space unnecessarily and should accommodate both large and small indices. It is difficult to come up with a set of hashing routines that can meet all these criteria, still maintain adequate efficiency, and not require a lengthy rehashing or reindexing operation. With additional work, extendible hashing could be made a viable alternative to B-trees for a file system.

Data Structure Summary

For file systems, the choice between hash tables and B-trees is an easy one. The problems that exist with hash tables present significant difficulties for a general-purpose indexing method when used as part of a file system. Resizing a hash table would potentially lock the entire file system for a long period of time while the table is resized and the elements rehashed, which is unacceptable for general use. B-trees, on the other hand, lend themselves very well to compact sizes when there are few keys, grow easily as the number of keys increases, and maintain a good search time (although not as good as hash tables). BFS uses B+trees for all of its indexing.

Connections: Indexing and the Rest of the File System

The most obvious questions to ask at this point are, How is the list of indices maintained? And where do individual indices live? That is, where do indices fit into the standard set of directories and files that exist on a file system? As with attributes, it is tempting to define new data structures for maintaining this information, but there is no need. BFS uses the normal directory structure to maintain the list of indices. BFS stores the data of each index in regular files that live in the index directory.

Although it is possible to put the index files into a user-visible directory with special protections, BFS instead stores the list of indices in a hidden directory created at file system creation time. The superblock stores the inode number of the index directory, which establishes the connection with the rest of the file system. The superblock is a convenient place to store hidden information such as this. Storing the indices in a hidden directory prevents accidental deletion of indices or other mishaps that could cause a catastrophic situation for the file system. The disadvantage of storing indices

in a hidden directory is that it requires a special-purpose API to access. This is the sort of decision that could go either way with little or no repercussions.

The API to operate on and access indices is simple. The operations that operate on entire indices are

- create index
- delete index
- open index directory
- read index directory
- stat index

It would be easy to extend this list of operations to support other common file operations (rename, etc.). But since there is little need for such operations on indices, BFS elects not to provide that functionality.

The create index operation simply takes an index name and the data type of the index. The name of the index connects the index with the corresponding attributes that will make use of the index. For example, the BeOS mail daemon adds an attribute named MAIL: from to all email messages it receives, and it also creates an index whose name is MAIL: from. The data type of the index should match the data type of the attributes. BFS supports the following data types for indices:

- String (up to 255 bytes)
- Integer (32-bit)
- Integer (64-bit)
- Float
- Double

Other types are certainly possible, but this set of data types covers the most general functionality. In practice almost all indices are string indices.

One "gotcha" when creating an index is that the name of an index may match files that already have that attribute. For example, if a file has an attribute named Foo and a program creates an index named Foo, the file that already had the attribute is not added to the newly created index. The difficulty is that there is no easy way to determine which files have the attribute without iterating over all files. Because creating indices is a relatively uncommon occurrence, it could be acceptable to iterate over all the files to find those that already have the attribute. BFS does not do this and pushes the responsibility onto the application developer. This deficiency of BFS is unfortunate, but there was no time in the development schedule to address it.

Deleting an index is a straightforward operation. Removing the file that contains the index from the index directory is all that is necessary. Although it is easy, deleting an index should be a rare operation since re-creating the index will not reindex all the files that have the attribute. For this reason an index should only be deleted when the only application that uses it is removed from the system and the index is empty (i.e., no files have the attribute).

The remaining index operations are simple housekeeping functions. The index directory functions (open, read, and close) allow a program to iterate over the index directory much like a program would iterate over a regular directory. The stat index function allows a program to check for the existence of an index and to obtain information about the size of the index. These routines all have trivial implementations since all the data structures involved are identical to that of regular directories and files.

Automatic Indices

In addition to allowing users to create their own indices, BFS supports built-in indices for the integral file attributes: name, size, and last modification. The file system itself must create and maintain these indices because it is the one that maintains those file attributes. Keep in mind that the name, size, and last modification time of a file are not regular attributes; they are integral parts of the i-node and not managed by the attribute code.

The name index keeps a list of all file names on the entire system. Every time a file name changes (creation, deletion, or rename), the file system must also update the name index. Adding a new file name to the name index happens after everything else about the file has been successfully created (i-node allocated and directory updated). The file name is then added to the name index. The insertion into the name index must happen as part of the file creation transaction so that should the system fail, the entire operation is undone as one transaction. Although it rarely happens, if the file name cannot be added to the name index (e.g., no space left), then the entire file creation must be undone.

Deletion of a file name is somewhat less problematic because it is unlikely to fail (no extra space is needed on the drive). Again though, deleting the name from the file name index should be the last operation done, and it should be done as part of the transaction that deletes the file so that the entire operation is atomic.

A rename operation is the trickiest operation to implement (in general and for the maintenance of the indices). As expected, updating the name index is the last thing done as part of the rename transaction. The rename operation itself decomposes into a deletion of the original name (if it exists) and an insertion of the new name into the index. Undoing a failure to insert the new name is particularly problematic. The rename operation may have deleted a file if the new name already existed (this is required for rename to be an atomic operation). However, because the other file is deleted (and its resources freed), undoing such an operation is extremely complex. Due to the complexity involved and the unlikelihood of the event even happening, BFS does not attempt to handle this case. Were the rename operation to be unable to insert the new name of a file into the name index, the file system would still be consistent, just not up-to-date (and the disk would most likely be 100% full as well).

Updates to the size index happen when a file changes size. As an optimization the file system only updates the size index when a file is closed. This prevents the file system from having to lock and modify the global size index for every write to any file. The disadvantage is that the size index may be slightly out-of-date with respect to certain files that are actively being written. The trade-off between being slightly out-of-date versus updating the size index on every write is well worth it—the performance hit is quite significant.

The other situation in which the size index can be a severe bottleneck is when there are many files of the same size. This may seem like an unusual situation, but it happens surprisingly often when running file system benchmarks that create and delete large numbers of files to test the speed of the file system. Having many files of the same size will stress the index structure and how it handles duplicate keys. BFS fares moderately well in this area, but performance degrades nonlinearly as the number of duplicates increases. Currently more than 10,000 or so duplicates causes the performance of modifications to the size index to lag noticeably.

The last modification time index is the final inherent file attribute that BFS indexes. Indexing the last modification time makes it easy for users to find recently created files or old files that are no longer needed. As expected, the last modification time index receives updates when a file is closed. The update consists of deleting the old last modification time and inserting a new time.

Knowing that an inherent index such as the last modification time index could be critical to system performance, BFS uses a slightly underhanded technique to improve the efficiency of the index. Since the last modification time has only 1-second granularity and it is possible to create many hundreds of files in 1 second, BFS scales the standard 32-bit time variable to 64 bits and adds in a small random component to reduce the potential number of duplicates. The random component is masked off when doing comparisons or passing the information to/from the user. In retrospect it would have been possible to use a 64-bit microsecond resolution timer and do similar masking of time values, but since the POSIX APIs only support 32-bit time values with 1-second resolution, there wasn't much point in defining a new, parallel set of APIs just to access a larger time value.

In addition to these three inherent file attributes, there are others that could also have been indexed. Early versions of BFS did in fact index the creation time of files, but we deemed this index to not be worth the performance penalty it cost. By eliminating the creation time index, the file system received roughly a 20% speed boost in a file create and delete benchmark. The trade-off is that it is not possible to use an index to search for files on their creation time, but we did not feel that this presented much of a loss. Similarly it would have been possible to index file access permissions, ownership information, and so on, but we chose not to because the cost of maintaining

the indices outweighed the benefit they would provide. Other file systems with different constraints might choose differently.

Other Attribute Indices

Aside from the inherent indices of name, size, and last modification time, there may be any number of other indices. Each of these indices corresponds to an attribute that programs store with files. As mentioned earlier, the BeOS mail system stores incoming email in individual files, tagging each file with attributes such as who the mail is from, who it is to, when it was sent, the subject, and so on. When first run, the mail system creates indices for each of the attributes that it writes. When the mail daemon writes one of these attributes to a file, the file system notices that the attribute name has a corresponding index and therefore updates the index as well as the file with the attribute value.

For every write to an attribute, the file system must also look in the index directory to see if the attribute name is the same as an index name. Although this may seem like it would slow the system down, the number of indices tends to be small (usually less than 100), and the cost of looking for an attribute is cheap since the data is almost always cached. When writing to an attribute, the file system also checks to see if the file already had the attribute. If so, it must delete the old value from the index first. Then the file system can add the new value to the file and insert the value into the corresponding attribute index. This all happens transparently to the user program.

When a user program deletes an attribute from a file, a similar set of operations happens. The file system must check if the attribute name being deleted has an index. If so, it must delete the attribute value from the index and then delete the attribute from the file.

The maintenance of indices complicates attribute processing but is necessary. The automatic management of indices frees programs from having to deal with the issue and offers a guarantee to programs that if an attribute index exists, the file system will keep it consistent with the state of all attributes written after the index is created.

BFS B+trees

BFS uses B+trees to store the contents of directories and all indexed information. The BFS B-tree implementation is a loose derivative of the B+trees described in the first edition Folk and Zoellick file structures textbook and owes a great deal to the public implementation of that data structure by Marcus J. Ranum. The B-tree code supports storing variable-sized keys along with a single disk offset (a 64-bit quantity in BFS). The keys stored in the tree can be strings, integers (32- and 64-bit), floats, or doubles. The biggest

departure from the original data structure was the addition of support for storing duplicate keys in the B+tree.

The API

The interface to the B+trees is also quite simple. The API has six main functions:

- Open/create a B+tree
- Insert a key/value pair
- Delete a key/value pair
- Find a key and return its value
- Go to the beginning/end of the tree
- Traverse the leaves of the tree (forwards/backwards)

The function that creates the B+tree has several parameters that allow specification of the node size of the B+tree, the data type to be stored in the tree, and various other bits of housekeeping information. The choice of node size for the B+tree is important. BFS uses a node size of 1024 bytes regardless of the block size of the file system. Determining the node size was a simple matter of experimentation and practicality. BFS supports file names up to 255 characters in length, which made a B+tree node size of 512 bytes too small. Larger B+trees tended to waste space because each node is never 100% full. This is particularly a problem for small directories. A size of 1024 bytes was chosen as a reasonable compromise.

The insertion routine accepts a key (whose type should match the data type of the B+tree), the length of the key, and a value. The value is a 64-bit i-node number that identifies which file corresponds to the key stored in the tree. If the key is a duplicate of an existing key and the tree does not allow duplicates, an error is returned. If the tree does support duplicates, the new value is inserted. In the case of duplicates, the value is used as a secondary key and must be unique (it is considered an error to insert the same key/value pair twice).

The delete routine takes a key/value pair as input and will search the tree for the key. If the key is found and it is not a duplicate, the key and its value are deleted from the tree. If the key is found and it has duplicate entries, the value passed in is searched for in the duplicates and that value removed.

The most basic operation is searching for a key in the B+tree. The find operation accepts an input key and returns the associated value. If the key contains duplicate entries, the first is returned.

The remaining functions support traversal of the tree so that a program can iterate over all the entries in the tree. It is possible to traverse the tree either forwards or backwards. That is, a forward traversal returns all the entries in ascending alphabetical or numerical order. A backwards traversal of the tree returns all the entries in descending order.

The Data Structure

The simplicity of the B+tree API belies the complexity of the underlying data structure. On disk, the B+tree is a collection of nodes. The very first node in all B+trees is a header node that contains a simple data structure that describes the rest of the B+tree. In essence it is a superblock for the B+tree. The structure is

```
long  magic;
int   node_size;
int   max_number_of_levels;
int   data_type;
off_t root_node_pointer;
off_t free_node_pointer;
off_t maximum_size;
```

The magic field is simply a magic number that identifies the block. Storing magic numbers like this aids in reconstructing file systems if corruption should occur. The next field, `node_size`, is the node size of the tree. Every node in the tree is always the same size (including the B+tree header node). The next field, `max_number_of_levels`, indicates how many levels deep the B+tree is. This depth of the tree is needed for various in-memory data structures. The data type field encodes the type of data stored in the tree (either 32-bit integers, 64-bit integers, floats, doubles, or strings).

The `root_node_pointer` field is the most important field. It contains the offset into the B+tree file of the root node of the tree. Without the address of the root node, it is impossible to use the tree. The root node must always be read to do any operation on a tree. The root node pointer, as with all disk offsets, is a 64-bit quantity.

The `free_node_pointer` field contains the address of the first free node in the tree. When deletions cause an entire node to become empty, the node is linked into a list that begins at this offset in the file. The list of free nodes is kept by linking the free nodes together. The link stored in each free node is simply the address of the next free node (and the last free node has a link address of -1).

The final field, `maximum_size`, records the maximum size of the B+tree file and is used to error-check node address requests. The `maximum_size` field is also used when requesting a new node and there are no free nodes. In that case the B+tree file is simply extended by writing to the end of the file. The address of the new node is the value of `maximum_size`. The `maximum_size` field is then incremented by the amount contained in the `node_size` variable.

The structure of interior and leaf nodes in the B+tree is the same. There is a short header followed by the packed key data, the lengths of the keys, and finally the associated values stored with each key. The header is enough to distinguish between leaf and interior nodes, and, as in all B+trees, only leaf nodes contain user data. The structure of nodes is

```

off_t  left link
off_t  right link
off_t  overflow link
short  count of keys in the node
short  length of all the keys
key data
short  key length index
off_t  array of the value for each key

```

The left and right links are used for leaf nodes to link them together so that it is easy to do an in-order traversal of the tree. The overflow link is used in interior nodes and refers to another node that effectively continues this node. The count of the keys in the node simply records how many keys exist in this node. The length of all the keys is added to the size of the header and then rounded up to a multiple of four to get to the beginning of the key length index. Each entry in the key length index stores the ending offset of the key (to compute the byte position in the node, the header size must also be added). That is, the first entry in the index contains the offset to the end of the first key. The length of a key can be computed by subtracting the previous entry's length (the first element's length is simply the value in the index). Following the length index is the array of key values (the value that was stored with the key). For interior nodes the value associated with a key is an offset to the corresponding node that contains elements less than this key. For leaf nodes the value associated with a key is the value passed by the user.

Duplicates

In addition to the interior and leaf nodes of the tree, there are also nodes that store the duplicates of a key. For reasons of efficiency, the handling of duplicates is rather complex. There are two types of duplicate nodes in the B-trees that BFS uses: duplicate fragment nodes and full duplicate nodes. A duplicate fragment node contains duplicates for several different keys. A full duplicate node stores duplicates for only one key.

The distinction between fragment node types exists because it is more common to have a small number of duplicates of a key than it is to have a large number of duplicates. That is, if there are several files with the same name in several different directories, it is likely that the number of duplicate names is less than eight. In fact, simple tests on a variety of systems reveal that as many as 35% of all file names are duplicates and have eight or fewer duplicates. Efficiently handling this case is important. Early versions of the BFS B-trees did not use duplicate fragments and we discovered that, when duplicating a directory hierarchy, a significant chunk of all the I/O being done was on behalf of handling duplicates in the name and size indices. By adding support for duplicate fragments, we were able to significantly re-

duce the amount of I/O that took place and sped up the time to duplicate a folder by nearly a factor of two.

When a duplicate entry must be inserted into a leaf node, instead of storing the user's value, the system stores a special value that is a pointer to either a fragment node or a full duplicate node. The value is special because it has its high bit(s) set. The BFS B-tree code reserves the top 2 bits of the value field to indicate if a value refers to duplicates. In general, this would not be acceptable, but because the file system only stores i-node numbers in the value field, we can be assured that this will not be a problem. Although this attitude has classically caused all sorts of headaches when a system grows, we are free from guilt in this instance. The safety of this approach stems from the fact that i-node numbers are disk block addresses, so they are at least 10 bits smaller than a raw disk byte address (because the minimum block size in BFS is 1024 bytes). Since the maximum disk size is 2^{64} bytes in BeOS and BFS uses a minimum of 1024-byte blocks, the maximum i-node number is 2^{54} . The value 2^{54} is small enough that it does not interfere with the top 2 bits used by the B-tree code.

When a duplicate key is inserted into a B-tree, the file system looks to see if any other keys in the current leaf node already have a duplicate fragment. If there is a duplicate fragment node that has space for another fragment, we insert our duplicate value into a new fragment within that node. If there are no other duplicate fragment nodes referenced in the current node, we create a new duplicate fragment node and insert the duplicate value there. If the key we're adding already has duplicates, we insert the duplicate into the fragment. If the fragment is full (it can only hold eight items), we allocate a full duplicate node and copy the existing duplicates into the new node. The full duplicate node contains space for more duplicates than a fragment, but there may still be more duplicates. To manage an arbitrary number of duplicates, full duplicate nodes contain links (forwards and backwards) to additional full duplicate pages. The list of duplicates is kept in sorted order based on the value associated with the key (i.e., the i-node number of the file that contains this key value as an attribute). This linear list of duplicates can become extremely slow to access when there are more than 10,000 or so duplicates. Unfortunately during the development of BFS there was not time to explore a better solution (such as storing another B-tree keyed on the i-node values).

Integration

In the abstract, the structure we have described has no connection to the rest of the file system; that is, it exists, but it is not clear how it integrates with the rest of the file system. The fundamental abstraction of BFS is an i-node that stores data. Everything is built up from this most basic abstraction. B-trees, which BFS uses to store directories and indices, are based on top of i-nodes. That is, the i-node manages the disk space allocated to the

B+tree, and the B+tree organizes the contents of that disk space into an index the rest of the system uses to look up information.

The B+trees use two routines, `read.data.stream()` and `write.data.stream()`, to access file data. These routines operate directly on i-nodes and provide the lowest level of access to file data in BFS. Despite their low-level nature, `read/write.data.stream()` have a very similar API to the higher-level `read()` and `write()` calls most programmers are familiar with. On top of this low-level I/O, the B+tree code implements the features discussed previously. The rest of the file system wraps around the B+tree functionality and uses it to provide directory and index abstractions. For example, creating a new directory involves creating a file and putting an empty B+tree into the file. When a program needs to enumerate the contents of a directory, the file system requests an in-order traversal of the B+tree. Opening a file contained in a directory is a lookup operation on the B+tree. The value returned by the lookup operation (if successful) is the i-node of the named file (which in turn is used to gain access to the file data). Creating a file inserts a new name/i-node pair into the B+tree. Likewise, deleting a file simply removes a name/i-node pair from a B+tree. Indices use the B+trees in much the same way as directories but allow duplicates where a directory does not.

5.3 Queries

If all the file system did with the indices was maintain them, they would be quite useless. The reason the file system bothers to manage indices is so that programs can issue queries that use the indices to efficiently obtain the results. The use of indices can speed up searches considerably over the brute-force alternative of examining every file in the file system.

In BFS, a query is simply a string that contains an expression about file attributes. The expression evaluates to true or false for any given file. If the expression is true for a file, then the file is in the result of the query. For example, the query

```
name == "main.c"
```

will only evaluate to true for files whose name is exactly `main.c`. The file system will evaluate this query by searching the name index to find files that match. Using the name index for this type of query is extremely efficient because it is a $\log(N)$ search on the name index B+tree instead of a linear search of all files. The difference in speed depends on the number of files on the file system, but for even a small system of 5000 files, the search time using the index is orders of magnitude faster than iterating over the files individually.

The result of a query is a list of files that match. The query API follows the POSIX directory iteration function API. There are three routines: open query, read query, and close query.

The open query routine accepts a string that represents the query and a flags argument that allows for any special options (such as live queries, which we will discuss later in this section). We will discuss the format of the query string next. The read query routine is called repeatedly, each time it returns the next file that matches the query until there are no more. When there are no more matching files, the read query routine returns an end-of-query indicator. The close query routine disposes of any resources and state associated with the query.

This simple API hides much of the complexity associated with processing queries. Query processing is the largest single chunk of code in BFS. Parsing queries, iterating over the parse trees, and deciding which files match a query requires a considerable amount of code. We now turn our attention to the details of that code.

Query Language

The query language that BFS supports is straightforward and very "C looking." While it would have been possible to use a more traditional database query language like SQL, it did not seem worth the effort. Because BFS is not a real database, we would have had considerable difficulty matching the semantics of SQL with the facilities of a file system. The BFS query language is built up out of simple expressions joined with logical AND or logical OR connectives. The grammar for a simple expression is

```
<attr-name> [logical-op] <value>
```

The `attr-name` is a simple text string that corresponds to the name of an attribute. The strings `MAIL:from`, `PERSON:email`, `name`, or `size` are all examples of valid `attr-names`. At least one of the attribute names in an expression must correspond to an index with the same name.

The `logical-op` component of the expression is one of the following operators:

- = (equality)
- != (inequality)
- < (less than)
- > (greater than)
- >= (greater than or equal to)
- <= (less than or equal to)

The value of an expression is a string. The string may be interpreted as a number if the data type of the attribute is numeric. If the `value` field is a string type, the `value` may be a regular expression (to allow wildcard matching).

These simple expressions may be grouped using logical AND (&&) or logical OR (||) connectives. Parentheses may also be used to group simple expressions and override the normal precedence of AND over OR. Finally, a logical

NOT may be applied to an entire expression by prefixing it with a "!" operator. The precedence of operators is the same as in the C programming language.

It is helpful to look at a few example queries to better understand the format. The first query we'll consider is

```
name == "*.c" && size > 20000
```

This query asks to find all files whose name is *.c (that is, ends with the characters .c) and whose size is greater than 20,000 bytes.

The query

```
(name == "*.c" || name == "*.h") && size > 20000
```

will find all files whose name ends in either .c or .h and whose size is greater than 20,000 bytes. The parentheses group the OR expression so that the AND conjunction [size > 20000] applies to both halves of the OR expression.

A final example demonstrates a fairly complex query:

```
(last_modified < 81793939 && size > 5000000) ||  
(name == "*.backup" && last_modified < 81793939)
```

This query asks to find all files last modified before a specific date and whose size is greater than 5 million bytes, OR all files whose name ends in .backup and who were last modified before a certain date. The date is expressed as the number of seconds since January 1, 1970 (i.e., it's in POSIX ctime format). This query would find very large files that have not been modified recently and backup files that have not been modified recently. Such a query would be useful for finding candidate files to erase or move to tape storage when trying to free up disk space on a full volume.

The query language BFS supports is rich enough to express almost any query about a set of files but yet still simple enough to be easily read and parsed.

Parsing Queries

The job of the BFS open query routine is to parse the query string (which also determines if it is valid) and to build a parse tree that represents the query. The parsing is done with a simple recursive descent parser (handwritten) that generates a tree as it parses through the query. If at any time the parser detects an error in the query string, it bubbles the error back to the top level and returns an error to the user. If the parse is successful, the resulting query tree is kept as part of the state associated with the object returned by the open query routine.

The parse tree that represents a query begins with a top-level node that maintains state about the entire query. From that node, pointers extend out to nodes representing AND and OR connectives. The leaves of the tree are

simple expressions that evaluate one value on a specific attribute. The leaves of the tree drive the evaluation of the query.

After parsing the query, the file system must decide how to evaluate the query. Deciding the evaluation strategy for the parse tree uses heuristics to walk the tree and find an optimal leaf node for beginning the evaluation. The heuristics BFS uses could, as always, stand some improvement. Starting at the root node, BFS attempts to walk down to a leaf node by picking a path that will result in the fewest number of matches. For example, in the query

```
name == "*.c" && size > 20000
```

there are two nodes, one that represents the left half [name == "*.c"] and one for the right half [size > 20000]. In choosing between these two expressions, the right half is a "tighter" expression because it is easier to evaluate than the left half. The left half of the query is more difficult to evaluate because it involves a regular expression. The use of a regular expression makes it impossible to take advantage of any fast searches of the name index since a B-tree is organized for exact matches. The right half of the query [size > 20000], on the other hand, can take advantage of the B-tree to find the first node whose size is 20,000 bytes and then to iterate in order over the remaining items in the tree (that are greater than the value 20,000).

The evaluation strategy also looks at the sizes of the indices to help it decide. If one index were significantly smaller in size than another, it makes more sense to iterate over the smaller index since it inherently will have fewer entries than the larger index. The logic controlling this evaluation is fairly convoluted. The complexity pays off though because picking the best path through a tree can result in significant savings in the time it takes to evaluate the query.

Read Query—The Real Work

The open query routine creates the parse tree and chooses an initial leaf node (i.e., query piece) to begin evaluation at. The real work of finding which files match the query is done by the read query routine. The read query routine begins iterating at the first leaf node chosen by the open query routine. Examining the leaf node, the read routine calls functions that know how to iterate through an index of a given data type and find files that match the leaf node expression.

Iterating through an index is complicated by the different types of logical operations that the query language supports. A less-than-or-equal comparison on a B-tree is slightly different than a less-than and is the inverse of a greater-than query. The number of logical comparisons (six) and the number of data types the file system supports (five) create a significant amount of similar but slightly different code.

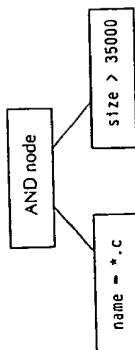


Figure 5-5 The parse tree for an AND query.

The process of iterating through all the values that match a particular query piece (e.g., a simple expression like `size < 500`) begins by finding the first matching item in the index associated with the query piece. In the case of an expression like `size < 500`, the iteration routine first finds the value 500, then traverses backward through the leaf items of the index B-tree to find the first value less than 500. If the traversal reaches the beginning of the tree, there are no items less than 500, and the iterator returns an error indicating that there are no more entries in this query piece. The iteration over all the matching items of one query piece is complicated because only one item is returned each iteration. This requires saving state between calls to be able to restart the search.

Once a matching file is found for a given query piece, the query engine must then travel back up the parse tree to see if the file matches the rest of the query. If the query in question was

```
name = *.c && size > 35000
```

then the resulting parse tree would be as shown in Figure 5-5.

The query engine would first descend down the right half of the parse tree because the `size > 35000` query piece is much less expensive to evaluate than the `name = *.c` half. For each file that matches the expression `size > 35000`, the query engine must also determine if it matches the expression `name = *.c`. Determining if a file matches the rest of the parse tree does not use other indices. The evaluation merely performs the comparison specified in each query piece directly against a particular file by reading the necessary attributes from the file.

The not-equal (!=) comparison operator presents an interesting difficulty for the query iterator. The interpretation of what "not equal" means is normally not open to discussion: either a particular value is not equal to another or it is. In the context of a query, however, it becomes less clear what the meaning is.

Consider the following query:

```
MAIL:status == New && MAIL:reply_to != mailinglist@noisy.com
```

This is a typical filter query used to only display all email not from a mailing list. The problem is that not all regular email messages will have a `Reply-To:` field in the message and thus will not have a `MAIL:reply_to` attribute. Even if

an email message does not have a `Reply-To:` field, it should still match the query. The original version of BFS required the attribute to be present for the file to match, which resulted in undesired behavior with email filters such as this.

To better support this style of querying, BFS changed its interpretation of the not-equal comparison. Now, if BFS encounters a not-equal comparison and the file in question does not have the attribute, then the file is still considered a match. This change in behavior complicates processing not-equal queries when the not-equal comparison is the only query piece. A query with a single query piece that has a not-equal comparison operator must now iterate through all files and cannot use any indexing to speed the search. All files that do not have the attribute will match the query, and those files that do have the attribute will only match if the value of the attribute is not equal to the value in the query piece. Although iterating over all files is dreadfully slow, it is necessary for the query engine to be consistent.

String Queries and Regular Expression Matching

By default, string matching in BFS is case-sensitive. This makes it easy to take advantage of the B-tree search routines, which are also case-sensitive. Queries that search for an exact string are extremely fast because this is exactly what B-trees were designed to do. Sadly, from a human interface standpoint, having to remember an exact file name, including the case of all the letters, is not acceptable. To allow more flexible searches, BFS supports string queries using regular expressions.

The regular expression matching supported by BFS is simple. The regular expression comparison function supports

- *—match any number of characters (including none)
- ?—match any single character
- [...]—match the range/class of characters in the []
- [...]—match the negated range/class of characters in the [^ ...]

The character class expressions allow matching specific ranges of characters. For example, all lowercase characters would be specified as `[a-z]`. The negated range expression, `[^]`, allows matching everything but that range of characters. For example, `[^0-9]` matches everything that is not a digit.

The typical query issued by the Tracker (the GUI file browser of the BeOS) is a case-insensitive substring query. That is, using the Tracker's find panel to search for the name "slow" translates into the following query:

```
name = "*[ss][l][o0][w]*"
```

Such a query must iterate through all the leaves of the name index and do a regular expression comparison on each name in the name index.

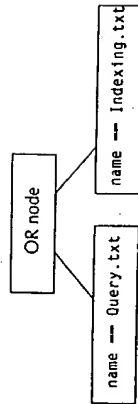


Figure 5-6 The parse tree for an OR query.

Unfortunately this obviates any benefit of B+trees and is much slower than doing a normal B+tree search. It is what end users expect, however, and that is more important than the use of an elegant B+tree search algorithm.

Additional Duties for Read Query

The read query routine also maintains additional state because it is repeatedly called to return results. The read query routine must be able to restart iterating over a query each time it is called. This requires saving the position in the query tree where the evaluation was as well as the position in the B+tree the query was iterating over.

Once a particular leaf node exhausts all the files in that index, the read query routine backs up the parse tree to see if it must descend down to another leaf node. In the following query:

```
name --- Query.txt || name --- Indexing.txt
```

the parse tree will have two leaves and will look like Figure 5-6.

The read query routine will iterate over the left half of the query, and when that exhausts all matches (most likely only one file), read query will back up to the OR node and descend down the right half of the tree. When the right half of the tree exhausts all matches, the query is done and read query returns its end-of-query indicator.

Once the query engine determines that a file matches a query, it must be returned to the program that called the read query routine. The result of a file match by the query engine is an i-node (recall that an index only stores the i-node number of a file in the index). The process of converting the result of a query into something appropriate for a user program requires the file system to convert an i-node into a file name. Normally this would not be possible, but BFS stores the name of a file (not the complete path, just the name) as an attribute of the file. Additionally, BFS stores a link in the file i-node to the directory that contains the file. This enables us to convert from an i-node address into a complete path to a file. It is quite unusual to store the name of a file in the file i-node, but BFS does this explicitly to support queries.

Live Queries

Live queries are another feature built around the query engine of BFS. A live query is a persistent query that monitors all file operations and reports additions to and deletions from the set of matching files. That is, if we issue the following as a live query:

```
name - *.c
```

the file system will first return to us all existing files whose name ends in .c. The live aspect of the query means that the file system will continue to inform us when any new files are created that match the query or when any existing files that matched are deleted or renamed. A more useful example of a live query is one that watches for new email. A live query with the predicate MAIL:status = New will monitor for newly arrived email and not require polling. A system administrator might wish to issue the live query size > 5000000 to monitor for files that are growing too large. Live queries reduce unnecessary polling in a system and do not lag behind the actual event as is common with polling.

To support this functionality the file system tags all indices it encounters when parsing the query. The tag associated with each index is a link back to the original parse tree of the query. Each time the file system modifies the index, it also traverses the list of live queries interested in modifications to the index and, for each, checks if the new file matches the query. Although this sounds deceptively simple, there were many subtle locking issues that needed to be dealt with properly to be able to traverse from indices to parse trees and then back again.

5.4 Summary

This lengthy chapter touched on numerous topics that relate to indexing in the Be File System. We saw that indices provide a mechanism for efficient access to all the files with a certain attribute. The name of an index corresponds to an attribute name. Whenever an attribute is written and its name matches an index, the file system also updates the index. The attribute index is keyed on the value written to the attribute, and the i-node address of the file is stored with the value. Storing the i-node address of the file that contains the attribute allows the file system to map from the entry in the index to the original file.

The file system maintains three indices that are inherent to a file (name, size, and last modification time). These indices require slightly special treatment because they are not real attributes in the same sense as attributes added by user programs. An index may or may not exist for other attributes added to a file.

We discussed several alternative approaches for the data structure of the index: B-trees, their variants, and hash tables. B-trees win out over hash tables because B-trees are more scalable and because there are no unexpected costly operations on B-trees like resizing a hash table.

The chapter then discussed the details of the BFS implementation of B-trees, their layout on disk, and how they handle duplicates. We observed that the management of duplicates in BFS is adequate, though perhaps not as high-performance as we would like. Then we briefly touched on how B-trees in BFS hook into the rest of the file system.

The final section discussed queries, covering what queries are, some of the parsing issues, how queries iterate over indices to generate results, and the way results are processed. The discussion also covered live queries and how they manage to send updates to a query when new files are created or when old files are deleted.

The substance of this chapter—attributes, indexing, and queries—is the essence of why BFS is interesting. The extensive use of these features in the BeOS is not seen in other systems.

6

Allocation Policies

6.1 Where Do You Put Things on Disk?

The Be File System views a disk as an array of blocks. The blocks are numbered beginning at zero and continuing up to the maximum disk block of the device. This view of a storage device is simple and easy to work with from a file system perspective. But the geometry of a physical disk is more than a simple array of disk blocks. The policies that the file system uses to arrange where data is on disk can have a significant impact on the overall performance of the file system. This chapter explains what allocation policies are, different ways to arrange data on disk, and other mechanisms for improving file system throughput by taking advantage of physical properties of disks.

6.2 What Are Allocation Policies?

An *allocation policy* is the set of rules and heuristics a file system uses to decide where to place items on a disk. The allocation policy dictates the location of file system metadata (i-nodes, directory data, and indices) as well as file data. The rules used for this task range from trivial to complex. Fortunately the effectiveness of a set of rules does not always match the complexity.

The goal of an allocation policy is to arrange data on disk so that the layout provides the best throughput possible when retrieving the data later. Several factors influence the success of an allocation policy. One key factor in defining good allocation policies is knowledge of how disks operate. Knowing what disks are good at and what operations are more costly can help when constructing an allocation policy.

HTTP Extensions for Distributed Authoring -- WEBDAV

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (1999). All Rights Reserved.

Abstract

This document specifies a set of methods, headers, and content-types ancillary to HTTP/1.1 for the management of resource properties, creation and management of resource collections, namespace manipulation, and resource locking (collision avoidance).

Contents

Status of this Memo	1
Abstract	1
Contents	2
1 Introduction	6
2 Notational Conventions	7
3 Terminology	7
4 Data Model for Resource Properties	8
4.1 The Resource Property Model	8
4.2 Existing Metadata Proposals	8
4.3 Properties and HTTP Headers	8
4.4 Property Values	9
4.5 Property Names	9
4.6 Media Independent Links	9
5 Collections of Web Resources	9
5.1 HTTP URL Namespace Model	10
5.2 Collection Resources	10
5.3 Creation and Retrieval of Collection Resources	11
5.4 Source Resources and Output Resources	11
6 Locking	12
6.1 Exclusive Vs. Shared Locks	12
6.2 Required Support	12
6.3 Lock Tokens	13
6.4 opaque/locked Lock Token URI Scheme	13
6.4.1 Node Field Generation Without the IEEE 802 Address	13
6.5 Lock Capability Discovery	14
6.6 Active Lock Discovery	15
6.7 Usage Considerations	15
7 Write Lock	16
7.1 Methods Restricted by Write Locks	16
7.2 Write Locks and Lock Tokens	16
7.3 Write Locks and Properties	16
7.4 Write Locks and Null Resources	16
7.5 Write Locks and Collections	16
7.6 Write Locks and the If Request Header	17
7.6.1 Example - Write Lock	17
7.7 Write Locks and COPY/MOVE	17
7.8 Refreshing Write Locks	18
8 HTTP Methods for Distributed Authoring	19
8.1 PROPFIND	19
8.1.1 Example - Retrieving Named Properties	20
8.1.2 Example - Using allprop to Retrieve All Properties	21
8.1.3 Example - Using proppatch to Retrieve all Property Names	23
8.2 PROPPATCH	24
8.2.1 Status Codes for use with 207 (Multi-Status)	24
8.2.2 Example - PROPPATCH	25
8.3 MKCOL Method	26
8.3.1 Request	26

9.7	Status-URL Response Header	43
9.8	Timeout Request Header	43
10	Status Code Extensions to HTTP/1.1	45
10.1	102 Processing	45
10.2	207 Multi-Status	45
10.3	422 Unprocessable Entity	45
10.4	423 Locked	45
10.5	424 Failed Dependency	45
10.6	507 Insufficient Storage	45
11	Multi-Status Response	46
12	XML Element Definitions	46
12.1	activelock XML Element	46
12.1.1	depth XML Element	46
12.1.2	locktoken XML Element	46
12.1.3	timeout XML Element	46
12.2	collection XML Element	47
12.3	href XML Element	47
12.4	link XML Element	47
12.4.1	dst XML Element	47
12.4.2	src XML Element	47
12.5	lockentry XML Element	47
12.6	lockinfo XML Element	48
12.7	lockscope XML Element	48
12.7.1	exclusive XML Element	48
12.7.2	shared XML Element	48
12.8	locktype XML Element	48
12.8.1	write XML Element	48
12.9	multistatus XML Element	49
12.9.1	response XML Element	49
12.9.2	responsedescription XML Element	49
12.10	owner XML Element	50
12.11	prop XML Element	50
12.12	propertybehavior XML element	50
12.12.1	keepalive XML element	50
12.12.2	omit XML element	51
12.13	propertyupdate XML element	51
12.13.1	remove XML element	51
12.13.2	set XML element	51
12.14	propfind XML Element	52
12.14.1	allprop XML Element	52
12.14.2	propname XML Element	52
13	DAV Properties	53
13.1	creationdate Property	53
13.2	displayname Property	53
13.3	getcontentlanguage Property	53
13.4	getcontentlength Property	53
13.5	getcontenttype Property	54
13.6	getetag Property	54
13.7	getlastmodified Property	54
13.8	lockdiscovery Property	54
13.8.1	Example - Retrieving the lockdiscovery Property	55
13.9	resourcetype Property	55
13.10	source Property	56
13.10.1	Example - A source Property	56
13.11	supportedlock Property	56
13.11.1	Example - Retrieving the supportedlock Property	57

8.3.2	Status Codes	26
8.3.3	Example - MKCOL	26
8.4	GET, HEAD for Collections	27
8.5	POST for Collections	27
8.6	DELETE	27
8.6.1	DELETE for Non-Collection Resources	27
8.6.2	DELETE for Collections	27
8.7	PUT	28
8.7.1	PUT for Non-Collection Resources	28
8.7.2	PUT for Collections	28
8.8	COPY Method	29
8.8.1	COPY for HTTP/1.1 resources	29
8.8.2	COPY for Properties	29
8.8.3	COPY for Collections	29
8.8.4	COPY and the Overwrite Header	30
8.8.5	Status Codes	30
8.8.6	Example - COPY with Overwrite	31
8.8.7	Example - COPY with No Overwrite	31
8.8.8	Example - COPY of a Collection	31
8.9	MOVE Method	32
8.9.1	MOVE for Properties	32
8.9.2	MOVE for Collections	32
8.9.3	MOVE and the Overwrite Header	33
8.9.4	Status Codes	33
8.9.5	Example - MOVE of a Non-Collection	33
8.9.6	Example - MOVE of a Collection	34
8.10	LOCK Method	34
8.10.1	Operation	34
8.10.2	The Effect of Locks on Properties and Collections	35
8.10.3	Locking Replicated Resources	35
8.10.4	Depth and Locking	35
8.10.5	Interaction with other Methods	35
8.10.6	Lock Compatibility Table	35
8.10.7	Status Codes	36
8.10.8	Example - Simple Lock Request	36
8.10.9	Example - Refreshing a Write Lock	37
8.10.10	Example - Multi-Resource Lock Request	38
8.11	UNLOCK Method	39
8.11.1	Example - UNLOCK	39
9	HTTP Headers for Distributed Authoring	40
9.1	DAV Header	40
9.2	Depth Header	40
9.3	Destination Header	41
9.4	If Header	41
9.4.1	No-tag-list Production	41
9.4.2	Tagged-list Production	41
9.4.3	not Production	42
9.4.4	Matching Function	42
9.4.5	If Header and Non-DAV Compliant Proxies	42
9.5	Lock-Token Header	43
9.6	Overwrite Header	43

14 Instructions for Processing XML in DAV	58
15 DAV Compliance Classes	58
15.1 Class 1	58
15.2 Class 2	58
16 Internationalization Considerations	59
17 Security Considerations	60
17.1 Authentication of Clients	60
17.2 Denial of Service	60
17.3 Security through Obscurity	60
17.4 Privacy Issues Connected to Locks	61
17.5 Privacy Issues Connected to Properties	61
17.6 Reduction of Security due to Source Link	61
17.7 Implications of XML External Entities	61
17.8 Risks Connected with Lock Tokens	61
18 IANA Considerations	62
19 Intellectual Property	63
20 Acknowledgements	63
21 References	64
21.1 Normative References	64
21.2 Informational References	65
22 Authors' Addresses	66
23 Appendices	67
23.1 Appendix 1 - WebDAV Document Type Definition	67
23.2 Appendix 2 - ISO 8601 Date and Time Profile	68
23.3 Appendix 3 - Notes on Processing XML Elements	69
23.3.1 Notes on Empty XML Elements	69
23.3.2 Notes on Illegal XML Processing	69
23.4 Appendix 4 - XML Namespaces for WebDAV	70
23.4.1 Introduction	70
23.4.2 Meaning of Qualified Names	70
24 Full Copyright Statement	71

1 Introduction

This document describes an extension to the HTTP/1.1 protocol that allows clients to perform remote web content authoring operations. This extension provides a coherent set of methods, headers, request entity body formats, and response entity body formats that provide operations for:

Properties: The ability to create, remove, and query information about Web pages, such as their authors, creation dates, etc. Also, the ability to link pages of any media type to related pages.

Collections: The ability to create sets of documents and to retrieve a hierarchical membership listing (like a directory listing in a file system).

Locking: The ability to keep more than one person from working on a document at the same time. This prevents the "lost update problem," in which modifications are lost as first one author then another writes changes without merging the other author's changes.

Namespace Operations: The ability to instruct the server to copy and move Web resources.

Requirements and rationale for these operations are described in a companion document, "Requirements for a Distributed Authoring and Versioning Protocol for the World Wide Web" [RFC2291].

The sections below provide a detailed introduction to resource properties (section 4), collections of resources (section 5), and locking operations (section 6). These sections introduce the abstractions manipulated by the WebDAV-specific HTTP methods described in section 8, "HTTP Methods for Distributed Authoring".

In HTTP/1.1, method parameter information was exclusively encoded in HTTP headers. Unlike HTTP/1.1, WebDAV encodes method parameter information either in an Extensible Markup Language (XML) [REC-XML] request entity body, or in an HTTP header. The use of XML to encode method parameters was motivated by the ability to add extra XML elements to existing structures, providing extensibility; and by XML's ability to encode information in ISO 10646 character sets, providing internationalization support. As a rule of thumb, parameters are encoded in XML entity bodies when they have unbounded length, or when they may be shown to a human user and hence require encoding in an ISO 10646 character set. Otherwise, parameters are encoded within HTTP headers. Section 9 describes the new HTTP headers used with WebDAV methods.

In addition to encoding method parameters, XML is used in WebDAV to encode the responses from methods, providing the extensibility and internationalization advantages of XML for method output, as well as input.

XML elements used in this specification are defined in section 12.

The XML namespace extension (Appendix 4) is also used in this specification in order to allow for new XML elements to be added without fear of colliding with other element names.

While the status codes provided by HTTP/1.1 are sufficient to describe most error conditions encountered by WebDAV methods, there are some errors that do not fall neatly into the existing categories. New status codes developed for the WebDAV methods are defined in section 10. Since some WebDAV methods may operate over many resources, the Multi-Status response has been introduced to return status information for multiple resources. The Multi-Status response is described in section 11.

WebDAV employs the property mechanism to store information about the current state of the resource. For example, when a lock is taken out on a resource, a lock information property describes the current state of the lock. Section 13 defines the properties used within the WebDAV specification.

Finishing off the specification are sections on what it means to be compliant with this specification (section 15), on internationalization support (section 16), and on security (section 17).

2 Notational Conventions

Since this document describes a set of extensions to the HTTP/1.1 protocol, the augmented BNF used herein to describe protocol elements is exactly the same as described in section 2.1 of [RFC2068]. Since this augmented BNF uses the basic production rules provided in section 2.2 of [RFC2068], these rules apply to this document as well.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

3 Terminology

URI/URL - A Uniform Resource Identifier and Uniform Resource Locator, respectively. These terms (and the distinction between them) are defined in [RFC2396].

Collection - A resource that contains a set of URIs, termed member URIs, which identify member resources and meets the requirements in section 5 of this specification.

Member URI - A URI which is a member of the set of URIs contained by a collection.

Internal Member URI - A Member URI that is immediately relative to the URI of the collection (the definition of immediately relative is given in section 5.2).

Property - A name/value pair that contains descriptive information about a resource.

Live Property - A property whose semantics and syntax are enforced by the server. For example, the live "getcontentlength" property has its value, the length of the entity returned by a GET request, automatically calculated by the server.

Dead Property - A property whose semantics and syntax are not enforced by the server. The server only records the value of a dead property; the client is responsible for maintaining the consistency of the syntax and semantics of a dead property.

Null Resource - A resource which responds with a 404 (Not Found) to any HTTP/1.1 or DAV method except for PUT, MKCOL, OPTIONS and LOCK. A NULL resource MUST NOT appear as a member of its parent collection.

4 Data Model for Resource Properties

4.1 The Resource Property Model

Properties are pieces of data that describe the state of a resource. Properties are data about data.

Properties are used in distributed authoring environments to provide for efficient discovery and management of resources. For example, a 'subject' property might allow for the indexing of all resources by their subject, and an 'author' property might allow for the discovery of what authors have written which documents.

The DAV property model consists of name/value pairs. The name of a property identifies the property's syntax and semantics, and provides an address by which to refer to its syntax and semantics.

There are two categories of properties: "live" and "dead". A live property has its syntax and semantics enforced by the server. Live properties include cases where a) the value of a property is read-only, maintained by the server, and b) the value of the property is maintained by the client, but the server performs syntax checking on submitted values. All instances of a given live property MUST comply with the definition associated with that property name. A dead property has its syntax and semantics enforced by the client; the server merely records the value of the property verbatim.

4.2 Existing Metadata Proposals

Properties have long played an essential role in the maintenance of large document repositories, and many current proposals contain some notion of a property, or discuss web metadata more generally. These include PICS [REC-PICS], PICS-NG, XML, Web Collections, and several proposals on representing relationships within HTML. Work on PICS-NG and Web Collections has been subsumed by the Resource Description Framework (RDF) metadata activity of the World Wide Web Consortium. RDF consists of a network-based data model and an XML representation of that model.

Some proposals come from a digital library perspective. These include the Dublin Core [RFC2413] metadata set and the Warwick Framework (WF), a container architecture for different metadata schemas. The literature includes many examples of metadata, including MARC [USMARC], a bibliographic metadata format, and a technical report bibliographic format employed by the Dienst system [RFC1807]. Additionally, the proceedings from the first IEEE Metadata conference describe many community-specific metadata sets.

Participants of the 1996 Metadata II Workshop in Warwick, UK [WF], noted that "new metadata sets will develop as the networked infrastructure matures" and "different communities will propose, design, and be responsible for different types of metadata." These observations can be corroborated by noting that many community-specific sets of metadata already exist, and there is significant motivation for the development of new forms of metadata as many communities increasingly make their data available in digital form, requiring a metadata format to assist data location and cataloging.

4.3 Properties and HTTP Headers

Properties already exist, in a limited sense, in HTTP message headers. However, in distributed authoring environments a relatively large number of properties are needed to describe the state of a resource, and setting/returning them all through HTTP headers is inefficient. Thus a mechanism is needed which allows a principal to identify a set of properties in which the principal is interested and to set or retrieve just those properties.

4.4 Property Values

The value of a property when expressed in XML MUST be well formed.

XML has been chosen because it is a flexible, self-describing, structured data format that supports rich schema definitions, and because of its support for multiple character sets. XML's self-describing nature allows any property's value to be extended by adding new elements. Older clients will not break when they encounter extensions because they will still have the data specified in the original schema and will ignore elements they do not understand. XML's support for multiple character sets allows any human-readable property to be encoded and read in a character set familiar to the user. XML's support for multiple human languages, using the "xml:lang" attribute, handles cases where the same character set is employed by multiple human languages.

4.5 Property Names

A property name is a universally unique identifier that is associated with a schema that provides information about the syntax and semantics of the property.

Because a property's name is universally unique, clients can depend upon consistent behavior for a particular property across multiple resources, on the same and across different servers, so long as that property is "live" on the resources in question, and the implementation of the live property is faithful to its definition.

The XML namespace mechanism, which is based on URIs [RFC2396], is used to name properties because it prevents namespace collisions and provides for varying degrees of administrative control.

The property namespace is flat; that is, no hierarchy of properties is explicitly recognized. Thus, if a property A and a property A/B exist on a resource, there is no recognition of any relationship between the two properties. It is expected that a separate specification will eventually be produced which will address issues relating to hierarchical properties.

Finally, it is not possible to define the same property twice on a single resource, as this would cause a collision in the resource's property namespace.

4.6 Media Independent Links

Although HTML resources support links to other resources, the Web needs more general support for links between resources of any media type (media types are also known as MIME types, or content types). WebDAV provides such links. A WebDAV link is a special type of property value, formally defined in section 12.4, that allows typed connections to be established between resources of any media type. The property value consists of source and destination Uniform Resource Identifiers (URIs); the property name identifies the link type.

5 Collections of Web Resources

This section provides a description of a new type of Web resource, the collection, and discusses its interactions with the HTTP URL namespace. The purpose of a collection resource is to model collection-like objects (e.g., file system directories) within a server's namespace.

All DAV compliant resources MUST support the HTTP URL namespace model specified herein.

5.1 HTTP URL Namespace Model

The HTTP URL namespace is a hierarchical namespace where the hierarchy is delimited with the "/" character.

An HTTP URL namespace is said to be consistent if it meets the following conditions: for every URL in the HTTP hierarchy there exists a collection that contains that URL as an internal member. The root, or top-level collection of the namespace under consideration is exempt from the previous rule.

Neither HTTP/1.1 nor WebDAV require that the entire HTTP URL namespace be consistent. However, certain WebDAV methods are prohibited from producing results that cause namespace inconsistencies.

Although implicit in [RFC2068] and [RFC2396], any resource, including collection resources, MAY be identified by more than one URI. For example, a resource could be identified by multiple HTTP URIs.

5.2 Collection Resources

A collection is a resource whose state consists of at least a list of internal member URIs and a set of properties, but which may have additional state such as entity bodies returned by GET. An internal member URI MUST be immediately relative to a base URI of the collection. That is, the internal member URI is equal to a containing collection's URI plus an additional segment for non-collection resources, or additional segment plus trailing slash "/" for collection resources, where segment is defined in section 3.3 of [RFC2396].

Any given internal member URI MUST only belong to the collection once, i.e., it is illegal to have multiple instances of the same URI in a collection. Properties defined on collections behave exactly as do properties on non-collection resources.

For all WebDAV compliant resources A and B, identified by URIs U and V, for which U is immediately relative to V, B MUST be a collection that has U as an internal member URI. So, if the resource with URL http://foo.com/bar/blah is WebDAV compliant and if the resource with URL http://foo.com/bar/ is WebDAV compliant then the resource with URL http://foo.com/bar/ must be a collection and must contain URL http://foo.com/bar/blah as an internal member.

Collection resources MAY list the URIs of non-WebDAV compliant children in the HTTP URL namespace hierarchy as internal members but are not required to do so. For example, if the resource with URL http://foo.com/bar/blah is not WebDAV compliant and the URL http://foo.com/bar/ identifies a collection then URL http://foo.com/bar/blah may or may not be an internal member of the collection with URL http://foo.com/bar/.

If a WebDAV compliant resource has no WebDAV compliant children in the HTTP URL namespace hierarchy then the WebDAV compliant resource is not required to be a collection.

There is a standing convention that when a collection is referred to by its name without a trailing slash, the trailing slash is automatically appended. Due to this, a resource may accept a URI without a trailing "/" to point to a collection. In this case it SHOULD return a content-location header in the response pointing to the URI ending with the "/". For example, if a client invokes a method on http://foo/bar/blah (no trailing slash), the resource http://foo/bar/blah (trailing slash) may respond as if the operation were invoked on it, and should return a content-location header with http://foo/bar/blah/ in it. In general clients SHOULD use the "/" form of collection names.

A resource MAY be a collection but not be WebDAV compliant. That is, the resource may comply with all the rules set out in this specification regarding how a collection is to behave without necessarily supporting all methods that a WebDAV compliant resource is required to support. In such a case the resource may return the DAV:resourcetype property with the value DAV:collection but MUST NOT return a DAV header containing the value "1" on an OPTIONS response.

5.3 Creation and Retrieval of Collection Resources

This document specifies the MKCOL method to create new collection resources, rather than using the existing HTTP/1.1 PUT or POST method, for the following reasons:

In HTTP/1.1, the PUT method is defined to store the request body at the location specified by the Request-URI. While a description format for a collection can readily be constructed for use with PUT, the implications of sending such a description to the server are undesirable. For example, if a description of a collection that omitted some existing resources were PUT to a server, this might be interpreted as a command to remove those members. This would extend PUT to perform DELETE functionality, which is undesirable since it changes the semantics of PUT, and makes it difficult to control DELETE functionality with an access control scheme based on methods.

While the POST method is sufficiently open-ended that a "create a collection" POST command could be constructed, this is undesirable because it would be difficult to separate access control for collection creation from other uses of POST.

The exact definition of the behavior of GET and PUT on collections is defined later in this document.

5.4 Source Resources and Output Resources

For many resources, the entity returned by a GET method exactly matches the persistent state of the resource, for example, a GIF file stored on a disk. For this simple case, the URI at which a resource is accessed is identical to the URI at which the source (the persistent state) of the resource is accessed. This is also the case for HTML source files that are not processed by the server prior to transmission.

However, the server can sometimes process HTML resources before they are transmitted as a return entity body. For example, a server-side-include directive within an HTML file might instruct a server to replace the directive with another value, such as the current date. In this case, what is returned by GET (HTML plus date) differs from the persistent state of the resource (HTML plus directive). Typically there is no way to access the HTML resource containing the unprocessed directive.

Sometimes the entity returned by GET is the output of a data-producing process that is described by one or more source resources (that may not even have a location in the URI namespace). A single data-producing process may dynamically generate the state of a potentially large number of output resources. An example of this is a CGI script that describes a "finger" gateway process that maps part of the namespace of a server into finger requests, such as http://www.foo.bar.org/finger_gateway/user@host.

In the absence of distributed authoring capabilities, it is acceptable to have no mapping of source resource(s) to the URI namespace. In fact, preventing access to the source resource(s) has desirable security benefits. However, if remote editing of the source resource(s) is desired, the source resource(s) should be given a location in the URI namespace. This source location should not be one of the locations at which the generated output is retrievable, since in general it is impossible for the server to differentiate requests for source resources from requests for process output resources. There is often a many-to-many relationship between source resources and output resources.

On WebDAV compliant servers the URI of the source resource(s) may be stored in a link on the output resource with type DAV:source (see section 13.10 for a description of the source link property). Storing the source URIs in links on the output resources places the burden of discovering the source on the authoring client. Note that the value of a source link is not guaranteed to point to the correct source. Source links may break or incorrect values may be entered. Also note that not all servers will allow the client to set the source link value. For example a server which generates source links on the fly for its CGI files will most likely not allow a client to set the source link value.

6 Locking

The ability to lock a resource provides a mechanism for serializing access to that resource. Using a lock, an authoring client can provide a reasonable guarantee that another principal will not modify a resource while it is being edited. In this way, a client can prevent the "lost update" problem.

This specification allows locks to vary over two client-specified parameters, the number of principals involved (exclusive vs. shared) and the type of access to be granted. This document defines locking for only one access type, write. However, the syntax is extensible, and permits the eventual specification of locking for other access types.

6.1 Exclusive Vs. Shared Locks

The most basic form of lock is an exclusive lock. This is a lock where the access right in question is only granted to a single principal. The need for this arbitration results from a desire to avoid having to merge results.

However, there are times when the goal of a lock is not to exclude others from exercising an access right but rather to provide a mechanism for principals to indicate that they intend to exercise their access rights. Shared locks are provided for this case. A shared lock allows multiple principals to receive a lock. Hence any principal with appropriate access can get the lock.

With shared locks there are two trust sets that affect a resource. The first trust set is created by access permissions. Principals who are trusted, for example, may have permission to write to the resource. Among those who have access permission to write to the resource, the set of principals who have taken out a shared lock also must trust each other, creating a (typically) smaller trust set within the access permission write set.

Starting with every possible principal on the Internet, in most situations the vast majority of these principals will not have write access to a given resource. Of the small number who do have write access, some principals may decide to guarantee their edits are free from overwrite conflicts by using exclusive write locks. Others may decide they trust their collaborators will not overwrite their work (the potential set of collaborators being the set of principals who have write permission) and use a shared lock, which informs their collaborators that a principal may be working on the resource.

The WebDAV extensions to HTTP do not need to provide all of the communications paths necessary for principals to coordinate their activities. When using shared locks, principals may use any out of band communication channel to coordinate their work (e.g., face-to-face interaction, written notes, post-it notes on the screen, telephone conversation, Email, etc.) The intent of a shared lock is to let collaborators know who else may be working on a resource.

Shared locks are included because experience from web distributed authoring systems has indicated that exclusive locks are often too rigid. An exclusive lock is used to enforce a particular editing process: take out an exclusive lock, read the resource, perform edits, write the resource, release the lock. This editing process has the problem that locks are not always properly released, for example when a program crashes, or when a lock owner leaves without unlocking a resource. While both timeouts and administrative action can be used to remove an offending lock, neither mechanism may be available when needed; the timeout may be long or the administrator may not be available.

6.2 Required Support

A WebDAV compliant server is not required to support locking in any form. If the server does support locking it may choose to support any combination of exclusive and shared locks for any access types.

The reason for this flexibility is that locking policy strikes to the very heart of the resource management and versioning systems employed by various storage repositories. These repositories

is the unicast/multicast bit, which will never be set in IEEE 802 addresses obtained from network cards; hence, there can never be a conflict between UUIDs generated by machines with and without network cards.

If a system does not have a primitive to generate cryptographic quality random numbers, then in most systems there are usually a fairly large number of sources of randomness available from which one can be generated. Such sources are system specific, but often include:

- the percent of memory in use
- the size of main memory in bytes
- the amount of free main memory in bytes
- the size of the paging or swap file in bytes
- free bytes of paging or swap file
- the total size of user virtual address space in bytes
- the total available user address space bytes
- the size of boot disk drive in bytes
- the free disk space on boot drive in bytes
- the current time
- the amount of time since the system booted
- the individual sizes of files in various system directories
- the creation, last read, and modification times of files in various system directories
- the utilization factors of various system resources (heap, etc.)
- current mouse cursor position
- current caret position
- current number of running processes, threads
- handles or IDs of the desktop window and the active window
- the value of stack pointer of the caller
- the process and thread ID of caller
- various processor architecture specific performance counters (instructions executed, cache misses, TLB misses)

(Note that it is precisely the above kinds of sources of randomness that are used to seed cryptographic quality random number generators on systems without special hardware for their construction.)

In addition, items such as the computer's name and the name of the operating system, while not strictly speaking random, will help differentiate the results from those obtained by other systems.

The exact algorithm to generate a node ID using these data is system specific, because both the data available and the functions to obtain them are often very system specific. However, assuming that one can concatenate all the values from the randomness sources into a buffer, and that a cryptographic hash function such as MD5 is available, then any 6 bytes of the MD5 hash of the buffer, with the multicast bit (the high bit of the first byte) set will be an appropriately random node ID.

Other hash functions, such as SHA-1, can also be used. The only requirement is that the result be suitably random_ in the sense that the outputs from a set uniformly distributed inputs are themselves uniformly distributed, and that a single bit change in the input can be expected to cause half of the output bits to change.

6.5 Lock Capability Discovery

Since server lock support is optional, a client trying to lock a resource on a server can either try the lock and hope for the best, or perform some form of discovery to determine what lock capabilities the server supports. This is known as lock capability discovery. Lock capability discovery differs from discovery of supported access control types, since there may be access control types without corresponding lock types. A client can determine what lock types the server supports by retrieving the supportedlock property.

require control over what sort of locking will be made available. For example, some repositories only support shared write locks while others only provide support for exclusive write locks while yet others use no locking at all. As each system is sufficiently different to merit exclusion of certain locking features, this specification leaves locking as the sole axis of negotiation within WebDAV.

6.3 Lock Tokens

A lock token is a type of state token, represented as a URI, which identifies a particular lock. A lock token is returned by every successful LOCK operation in the lockdiscovery property in the response body, and can also be found through lock discovery on a resource.

Lock token URIs MUST be unique across all resources for all time. This uniqueness constraint allows lock tokens to be submitted across resources and servers without fear of confusion.

This specification provides a lock token URI scheme called opaquelocktoken that meets the uniqueness requirements. However resources are free to return any URI scheme so long as it meets the uniqueness requirements.

Having a lock token provides no special access rights. Anyone can find out anyone else's lock token by performing lock discovery. Locks MUST be enforced based upon whatever authentication mechanism is used by the server, not based on the secrecy of the token values.

6.4 opaquelocktoken Lock Token URI Scheme

The opaquelocktoken URI scheme is designed to be unique across all resources for all time. Due to this uniqueness quality, a client may submit an opaque lock token in an If header on a resource other than the one that returned it.

All resources MUST recognize the opaquelocktoken scheme and, at minimum, recognize that the lock token does not refer to an outstanding lock on the resource.

In order to guarantee uniqueness across all resources for all time the opaquelocktoken requires the use of the Universal Unique Identifier (UUID) mechanism, as described in [ISO-11578].

Opaquelocktoken generators, however, have a choice of how they create these tokens. They can either generate a new UUID for every lock token they create or they can create a single UUID and then add extension characters. If the second method is selected then the program generating the extensions MUST guarantee that the same extension will never be used twice with the associated UUID.

OpaqueLockToken-URI = "opaquelocktoken:" UUID [Extension] ; The UUID production is the string representation of a UUID, as defined in [ISO-11578]. Note that white space (WS) is not allowed between elements of this production.

Extension = path ; path is defined in section 3.2.1 of RFC 2068 [RFC2068]

6.4.1 Node Field Generation Without the IEEE 802 Address

UUIDs, as defined in [ISO-11578], contain a "node" field that contains one of the IEEE 802 addresses for the server machine. As noted in section 17.8, there are several security risks associated with exposing a machine's IEEE 802 address. This section provides an alternate mechanism for generating the "node" field of a UUID which does not employ an IEEE 802 address. WebDAV servers MAY use this algorithm for creating the node field when generating UUIDs. The text in this section is originally from an Internet-Draft by Paul Leach and Rich Salz, who are noted here to properly attribute their work.

The ideal solution is to obtain a 47 bit cryptographic quality random number, and use it as the low 47 bits of the node ID, with the most significant bit of the first octet of the node ID set to 1. This bit

Any DAV compliant resource that supports the LOCK method MUST support the supportedlock property.

6.6 Active Lock Discovery

If another principal locks a resource that a principal wishes to access, it is useful for the second principal to be able to find out who the first principal is. For this purpose the lockdiscovery property is provided. This property lists all outstanding locks, describes their type, and where available, provides their lock token.

Any DAV compliant resource that supports the LOCK method MUST support the lockdiscovery property.

6.7 Usage Considerations

Although the locking mechanisms specified here provide some help in preventing lost updates, they cannot guarantee that updates will never be lost. Consider the following scenario:

Two clients A and B are interested in editing the resource 'index.html'. Client A is an HTTP client rather than a WebDAV client, and so does not know how to perform locking.

Client A doesn't lock the document, but does a GET and begins editing.

Client B does LOCK, performs a GET and begins editing.

Client B finishes editing, performs a PUT, then an UNLOCK.

Client A performs a PUT, overwriting and losing all of B's changes.

There are several reasons why the WebDAV protocol itself cannot prevent this situation. First, it cannot force all clients to use locking because it must be compatible with HTTP clients that do not comprehend locking. Second, it cannot require servers to support locking because of the variety of repository implementations, some of which rely on reservations and merging rather than on locking. Finally, being stateless, it cannot enforce a sequence of operations like LOCK / GET / PUT / UNLOCK.

WebDAV servers that support locking can reduce the likelihood that clients will accidentally overwrite each other's changes by requiring clients to lock resources before modifying them. Such servers would effectively prevent HTTP 1.0 and HTTP 1.1 clients from modifying resources.

WebDAV clients can be good citizens by using a lock / retrieve / write /unlock sequence of operations (at least by default) whenever they interact with a WebDAV server that supports locking.

HTTP 1.1 clients can be good citizens, avoiding overwriting other clients' changes, by using entity tags in If-Match headers with any requests that would modify resources.

Information managers may attempt to prevent overwrites by implementing client-side procedures requiring locking before modifying WebDAV resources.

7 Write Lock

This section describes the semantics specific to the write lock type. The write lock is a specific instance of a lock type, and is the only lock type described in this specification.

7.1 Methods Restricted by Write Locks

A write lock MUST prevent a principal without the lock from successfully executing a PUT, POST, PROPPATCH, LOCK, UNLOCK, MOVE, DELETE, or MKCOL on the locked resource. All other current methods, GET in particular, function independently of the lock.

Note, however, that as new methods are created it will be necessary to specify how they interact with a write lock.

7.2 Write Locks and Lock Tokens

A successful request for an exclusive or shared write lock MUST result in the generation of a unique lock token associated with the requesting principal. Thus if five principals have a shared write lock on the same resource there will be five lock tokens, one for each principal.

7.3 Write Locks and Properties

While those without a write lock may not alter a property on a resource it is still possible for the values of live properties to change, even while locked, due to the requirements of their schemas. Only dead properties and live properties defined to respect locks are guaranteed not to change while write locked.

7.4 Write Locks and Null Resources

It is possible to assert a write lock on a null resource in order to lock the name.

A write locked null resource, referred to as a lock-null resource, MUST respond with a 404 (Not Found) or 405 (Method Not Allowed) to any HTTP/1.1 or DAV methods except for PUT, MKCOL, OPTIONS, PROPFIND, LOCK, and UNLOCK. A lock-null resource MUST appear as a member of its parent collection. Additionally the lock-null resource MUST have defined on it all mandatory DAV properties. Most of these properties, such as all the get* properties, will have no value as a lock-null resource does not support the GET method. Lock-Null resources MUST have defined values for lockdiscovery and supportedlock properties.

Until a method such as PUT or MKCOL is successfully executed on the lock-null resource the resource MUST stay in the lock-null state. However, once a PUT or MKCOL is successfully executed on a lock-null resource the resource ceases to be in the lock-null state.

If the resource is unlocked, for any reason, without a PUT, MKCOL, or similar method having been successfully executed upon it then the resource MUST return to the null state.

7.5 Write Locks and Collections

A write lock on a collection, whether created by a "Depth: 0" or "Depth: infinity" lock request, prevents the addition or removal of member URIs of the collection by non-lock owners. As a consequence, when a principal issues a PUT or POST request to create a new resource under a URI which needs to be an internal member of a write locked collection to maintain HTTP namespace consistency, or issues a DELETE to remove a resource which has a URI which is an existing internal member URI of a write locked collection, this request MUST fail if the principal does not have a write lock on the collection.

However, if a write lock request is issued to a collection containing member URIs identifying resources that are currently locked in a manner which conflicts with the write lock, the request MUST fail with a 423 (Locked) status code.

If a lock owner causes the URI of a resource to be added as an internal member URI of a locked collection then the new resource MUST be automatically added to the lock. This is the only mechanism that allows a resource to be added to a write lock. Thus, for example, if the collection */a/b/* is write locked and the resource */c* is moved to */a/b/c* then resource */a/b/c* will be added to the write lock.

7.6 Write Locks and the If Request Header

If a user agent is not required to have knowledge about a lock when requesting an operation on a locked resource, the following scenario might occur. Program A, run by User A, takes out a write lock on a resource. Program B, also run by User A, has no knowledge of the lock taken out by Program A, yet performs a PUT to the locked resource. In this scenario, the PUT succeeds because locks are associated with a principal, not a program, and thus program B, because it is acting with principal A's credential, is allowed to perform the PUT. However, had program B known about the lock, it would not have overwritten the resource, preferring instead to present a dialog box describing the conflict to the user. Due to this scenario, a mechanism is needed to prevent different programs from accidentally ignoring locks taken out by other programs with the same authorization.

In order to prevent these collisions a lock token MUST be submitted by an authorized principal in the If header for all locked resources that a method may interact with or the method MUST fail. For example, if a resource is to be moved and both the source and destination are locked then two lock tokens must be submitted, one for the source and the other for the destination.

7.6.1 Example - Write Lock

>>>Request

```
COPY /~f/fielding/index.html HTTP/1.1
Host: www.ics.uci.edu
Destination: http://www.ics.uci.edu/users/f/fielding/index.html
If: <http://www.ics.uci.edu/users/f/fielding/index.html>
    <opaquelocktoken:f81d4fae-7dec-11d0-a765-00a0c91e6bf6>
```

>>>Response

HTTP/1.1 204 No Content

In this example, even though both the source and destination are locked, only one lock token must be submitted, for the lock on the destination. This is because the source resource is not modified by a COPY, and hence unaffected by the write lock. In this example, user agent authentication has previously occurred via a mechanism outside the scope of the HTTP protocol, in the underlying transport layer.

7.7 Write Locks and COPY/MOVE

A COPY method invocation MUST NOT duplicate any write locks active on the source. However, as previously noted, if the COPY copies the resource into a collection that is locked with "Depth: infinity", then the resource will be added to the lock.

A successful MOVE request on a write locked resource MUST NOT move the write lock with the resource. However, the resource is subject to being added to an existing lock at the destination, as specified in section 7.5. For example, if the MOVE makes the resource a child of a collection that is locked with "Depth: infinity", then the resource will be added to that collection's lock. Additionally,

if a resource locked with "Depth: infinity" is moved to a destination that is within the scope of the same lock (e.g., within the namespace tree covered by the lock), the moved resource will again be added to the lock. In both these examples, as specified in section 7.6, an If header must be submitted containing a lock token for both the source and destination.

7.8 Refreshing Write Locks

A client MUST NOT submit the same write lock request twice. Note that a client is always aware it is resubmitting the same lock request because it must include the lock token in the If header in order to make the request for a resource that is already locked.

However, a client may submit a LOCK method with an If header but without a body. This form of LOCK MUST only be used to "refresh" a lock. Meaning, at minimum, that any timers associated with the lock MUST be re-set.

A server may return a Timeout header with a lock refresh that is different than the Timeout header returned when the lock was originally requested. Additionally clients may submit Timeout headers of arbitrary value with their lock refresh requests. Servers, as always, may ignore Timeout headers submitted by the client.

If an error is received in response to a refresh LOCK request the client SHOULD assume that the lock was not refreshed.

8 HTTP Methods for Distributed Authoring

The following new HTTP methods use XML as a request and response format. All DAV compliant clients and resources MUST use XML parsers that are compliant with [REC-XML]. All XML used in either requests or responses MUST be, at minimum, well formed. If a server receives ill-formed XML in a request it MUST reject the entire request with a 400 (Bad Request). If a client receives ill-formed XML in a response then it MUST NOT assume anything about the outcome of the executed method and SHOULD treat the server as malfunctioning.

8.1 PROPFIND

The PROPFIND method retrieves properties defined on the resource identified by the Request-URL, if the resource does not have any internal members, or on the resource identified by the Request-URL and potentially its member resources, if the resource is a collection that has internal member URLs. All DAV compliant resources MUST support the PROPFIND method and the proppfind XML element (section 12.14) along with all XML elements defined for use with that element.

A client may submit a Depth header with a value of "0", "1", or "infinity" with a PROPFIND on a collection resource with internal member URLs. DAV compliant servers MUST support the "0", "1" and "infinity" behaviors. By default, the PROPFIND method without a Depth header MUST act as if a "Depth: infinity" header was included.

A client may submit a proppfind XML element in the body of the request method describing what information is being requested. It is possible to request particular property values, all property values, or a list of the names of the resource's properties. A client may choose not to submit a request body. An empty PROPFIND request body MUST be treated as a request for the names and values of all properties.

All servers MUST support returning a response of content type text/xml or application/xml that contains a multistatus XML element that describes the results of the attempts to retrieve the various properties.

If there is an error retrieving a property then a proper error result MUST be included in the response. A request to retrieve the value of a property which does not exist is an error and MUST be noted, if the response uses a multistatus XML element, with a response XML element which contains a 404 (Not Found) status value.

Consequently, the multistatus XML element for a collection resource with member URIs MUST include a response XML element for each member URI of the collection, to whatever depth was requested. Each response XML element MUST contain an href XML element that gives the URI of the resource on which the properties in the prop XML element are defined. Results for a PROPFIND on a collection resource with internal member URIs are returned as a flat list whose order of entries is not significant.

In the case of allprop and proppname, if a principal does not have the right to know whether a particular property exists then the property should be silently excluded from the response.

The results of this method SHOULD NOT be cached.

8.1.1 Example - Retrieving Named Properties

>>Request

```
PROPFIND /file HTTP/1.1
Host: www.foo.bar
Content-type: text/xml; charset=utf-8
Content-Length: xxxxx

<?xml version="1.0" encoding="utf-8" ?>
<D:proppfind xmlns:D="DAV:">
  <R:bigbox/>
  <R:author/>
  <R:Disallowing/>
  <R:Random/>
  <D:prop>
</D:prop>
</D:proppfind>
```

>>Response

```
HTTP/1.1 207 Multi-Status
Content-Type: text/xml; charset=utf-8
Content-Length: xxxxx

<?xml version="1.0" encoding="utf-8" ?>
<D:multistatus xmlns:D="DAV:">
  <D:response>
    <D:href>http://www.foo.bar/file</D:href>
    <D:propstat>
      <D:prop xmlns:R="http://www.foo.bar/boxschema/">
        <R:bigbox>
          <R:BoxType>Box type A</R:BoxType>
        </R:bigbox>
        <R:author>
          <R:Name>J. J. Johnson</R:Name>
        </R:author>
      </D:prop>
      <D:status>HTTP/1.1 200 OK</D:status>
    </D:propstat>
    <D:propstat>
      <D:prop><R:Disallowing/><R:Random/></D:prop>
      <D:status>HTTP/1.1 403 Forbidden</D:status>
    </D:propstat>
    <D:response>
      <D:response><D:response> The user does not have
      access to the Disallowing property.
      </D:propstat>
    </D:response>
    <D:response>
      <D:response><D:response> There has been an access violation
      error. </D:response>
    </D:multistatus>
```

In this example, PROPFIND is executed on a non-collection resource <http://www.foo.bar/file>. The proppfind XML element specifies the name of four properties whose values are being requested. In this case only two properties were returned, since the principal issuing the request did not have sufficient access rights to see the third and fourth properties.

8.1.2 Example - Using allprop to Retrieve All Properties

>>Request

```
PROPFIND /container/ HTTP/1.1
Host: www.foo.bar
Depth: 1
Content-Type: text/xml; charset=utf-8
Content-Length: xxxx

<?xml version="1.0" encoding="utf-8" ?>
<D:propfind xmlns:D="DAV:">
  <D:allprop/>
</D:propfind>
```

>>Response

```
HTTP/1.1 207 Multi-Status
Content-Type: text/xml; charset=utf-8
Content-Length: xxxx

<?xml version="1.0" encoding="utf-8" ?>
<D:multistatus xmlns:D="DAV:">
  <D:response>
    <D:href>http://www.foo.bar/container/</D:href>
    <D:propstat>
      <D:prop xmlns:D="http://www.foo.bar/boxschema/">
        <R:bigbox>
          <R:BoxType>Box Type A</R:BoxType>
          <R:bigbox>
            <R:author>
              <R:Name>Hadrian</R:Name>
            </R:author>
            <D:creationdate>
              1997-12-01T17:42:21-08:00
            </D:creationdate>
            <D:displayname>
              Example collection
            </D:displayname>
            <D:resourceType><D:collection/></D:resourceType>
            <D:supportedlock>
              <D:lockentry>
                <D:lockscope><D:exclusive/></D:lockscope>
                <D:locktype><D:write/></D:locktype>
              </D:lockentry>
              <D:lockentry>
                <D:lockscope><D:shared/></D:lockscope>
                <D:locktype><D:write/></D:locktype>
              </D:lockentry>
            </D:supportedlock>
          </D:prop>
          <D:status>HTTP/1.1 200 OK</D:status>
        </D:propstat>
      </D:response>
    <D:href>http://www.foo.bar/container/front.html</D:href>
    <D:propstat>
      <D:prop xmlns:D="http://www.foo.bar/boxschema/">
        <R:bigbox>
          <R:BoxType>Box Type B</R:BoxType>
          <R:bigbox>
            <D:creationdate>
              1997-12-01T18:27:21-08:00
            </D:creationdate>
            <D:displayname>
              Example HTML resource
            </D:displayname>
```

```
<D:getcontentlength>
  4525
</D:getcontentlength>
<D:getcontenttype>
  text/html
</D:getcontenttype>
<D:getetag>
  zzyzx
</D:getetag>
<D:getlastmodified>
  Monday, 12-Jan-98 09:25:56 GMT
</D:getlastmodified>
<D:resourceType/>
<D:supportedlock>
  <D:lockentry>
    <D:lockscope><D:exclusive/></D:lockscope>
    <D:locktype><D:write/></D:locktype>
  </D:lockentry>
  <D:lockentry>
    <D:lockscope><D:shared/></D:lockscope>
    <D:locktype><D:write/></D:locktype>
  </D:lockentry>
</D:supportedlock>
<D:prop>
  <D:status>HTTP/1.1 200 OK</D:status>
  <D:propstat>
    <D:response>
      <D:multistatus>
```

In this example, PROPFIND was invoked on the resource `http://www.foo.bar/container/` with a Depth header of 1, meaning the request applies to the resource and its children, and a propfind XML element containing the allprop XML element, meaning the request should return the name and value of all properties defined on each resource.

The resource `http://www.foo.bar/container/` has six properties defined on it:

`http://www.foo.bar/boxschema/bigbox`, `http://www.foo.bar/boxschema/author`, `DAV:creationdate`, `DAV:displayname`, `DAV:resourceType`, and `DAV:supportedlock`.

The last four properties are WebDAV-specific, defined in section 13. Since GET is not supported on this resource, the get* properties (e.g., `getcontentlength`) are not defined on this resource. The DAV-specific properties assert that "container" was created on December 1, 1997, at 5:42:21PM, in a time zone 8 hours west of GMT (creationdate), has a name of "Example collection" (displayname), a collection resource type (resourceType), and supports exclusive write and shared write locks (supportedlock).

The resource `http://www.foo.bar/container/front.html` has nine properties defined on it:

`http://www.foo.bar/boxschema/bigbox` (another instance of the "bigbox" property type), `DAV:creationdate`, `DAV:displayname`, `DAV:getcontentlength`, `DAV:getcontenttype`, `DAV:getetag`, `DAV:getlastmodified`, `DAV:resourceType`, and `DAV:supportedlock`.

The DAV-specific properties assert that "front.html" was created on December 1, 1997, at 6:27:21PM, in a time zone 8 hours west of GMT (creationdate), has a name of "Example HTML resource" (displayname), a content length of 4525 bytes (getcontentlength), a MIME type of "text/html" (getcontenttype), an entity tag of "zzyzx" (getetag), was last modified on Monday, January 12, 1998, at 09:25:56 GMT (getlastmodified), has an empty resource type, meaning that it is not a collection (resourceType), and supports both exclusive write and shared write locks (supportedlock).

8.1.3 Example - Using propname to Retrieve all Property Names

```
>>Request

PROPFIND /container/ HTTP/1.1
Host: www.foo.bar
Content-Type: text/xml; charset=utf-8
Content-Length: xxxx

<?xml version="1.0" encoding="utf-8" ?>
<propfind xmlns="DAV:">
  <propname/>
</propfind>

>>Response

HTTP/1.1 207 Multi-Status
Content-Type: text/xml; charset=utf-8
Content-Length: xxxx

<?xml version="1.0" encoding="utf-8" ?>
<multistatus xmlns="DAV:">
  <response>
    <href>http://www.foo.bar/container/</href>
    <propstat>
      <prop xmlns="R:">http://www.foo.bar/boxschema/">
        <R:bigbox/>
        <R:author/>
        <R:creationdate/>
        <R:displayname/>
        <R:resource-type/>
        <R:supportedlock/>
      </prop>
      <status>HTTP/1.1 200 OK</status>
    </propstat>
  </response>
  <response>
    <href>http://www.foo.bar/container/front.html</href>
    <propstat>
      <prop xmlns="R:">http://www.foo.bar/boxschema/">
        <R:bigbox/>
        <R:creationdate/>
        <R:displayname/>
        <R:getcontentlength/>
        <R:getcontenttype/>
        <R:getetag/>
        <R:getlastmodified/>
        <R:resource-type/>
        <R:supportedlock/>
      </prop>
      <status>HTTP/1.1 200 OK</status>
    </propstat>
  </response>
</multistatus>
```

In this example, PROPFIND is invoked on the collection resource `http://www.foo.bar/container/`, with a `propfind` XML element containing the `propname` XML element, meaning the name of all properties should be returned. Since no Depth header is present, it assumes its default value of "infinity", meaning the name of the properties on the collection and all its progeny should be returned.

Consistent with the previous example, resource `http://www.foo.bar/container/` has six properties defined on it, `http://www.foo.bar/boxschema/bigbox`, `http://www.foo.bar/boxschema/author`, `DAV:creationdate`, `DAV:displayname`, `DAV:resource-type`, and `DAV:supportedlock`.

The resource `http://www.foo.bar/container/index.html`, a member of the "container" collection, has nine properties defined on it, `http://www.foo.bar/boxschema/bigbox`, `DAV:creationdate`, `DAV:displayname`, `DAV:getcontentlength`, `DAV:resource-type`, `DAV:etag`, `DAV:getlastmodified`, `DAV:resource-type`, and `DAV:supportedlock`.

This example also demonstrates the use of XML namespace scoping, and the default namespace. Since the "xmlns" attribute does not contain an explicit "short-hand name" (prefix) letter, the namespace applies by default to all enclosed elements. Hence, all elements which do not explicitly state the namespace to which they belong are members of the "DAV:" namespace schema.

8.2 PROPPATCH

The PROPPATCH method processes instructions specified in the request body to set and/or remove properties defined on the resource identified by the Request-URI.

All DAV compliant resources MUST support the PROPPATCH method and MUST process instructions that are specified using the `propertyupdate`, `set`, and `remove` XML elements of the DAV schema. Execution of the directives in this method is, of course, subject to access control constraints. DAV compliant resources SHOULD support the setting of arbitrary dead properties.

The request message body of a PROPPATCH method MUST contain the `propertyupdate` XML element. Instruction processing MUST occur in the order instructions are received (i.e., from top to bottom). Instructions MUST either all be executed or none executed. Thus if any error occurs during processing all executed instructions MUST be undone and a proper error result returned. Instruction processing details can be found in the definition of the `set` and `remove` instructions in section 12.13.

8.2.1 Status Codes for use with 207 (Multi-Status)

The following are examples of response codes one would expect to be used in a 207 (Multi-Status) response for this method. Note, however, that unless explicitly prohibited any 207 (Multi-Status) response code may be used in a 207 (Multi-Status) response.

- 200 (OK) - The command succeeded. As there can be a mixture of sets and removes in a body, a 201 (Created) seems inappropriate.
- 403 (Forbidden) - The client, for reasons the server chooses not to specify, cannot alter one of the properties.
- 409 (Conflict) - The client has provided a value whose semantics are not appropriate for the property. This includes trying to set read-only properties.
- 423 (Locked) - The specified resource is locked and the client either is not a lock owner or the lock type requires a lock token to be submitted and the client did not submit it.
- 507 (Insufficient Storage) - The server did not have sufficient space to record the property.

8.2.2 Example - PROPPATCH

>>Request

```
PROPPATCH /bar.html HTTP/1.1
Host: www.foo.com
Content-Type: text/xml; charset=utf-8
Content-Length: xxxx

<?xml version="1.0" encoding="utf-8" ?>
<D:propertyupdate xmlns:D="DAV:"
  xmlns:Z="http://www.w3.com/standards/z39.50/">
  <D:set>
    <D:prop>
      <Z:author>
        <Z:Author>Jim Whitehead</Z:Author>
        <Z:Author>Roy Fielding</Z:Author>
      </Z:author>
    </D:prop>
    <D:set>
      <D:remove>
        <D:prop><Z:Copyright-Owner/></D:prop>
      </D:remove>
    </D:propertyupdate>
  </D:propertyupdate>
</D:propertyupdate>
```

>>Response

```
HTTP/1.1 207 Multi-Status
Content-Type: text/xml; charset=utf-8
Content-Length: xxxx

<?xml version="1.0" encoding="utf-8" ?>
<D:multistatus xmlns:D="DAV:"
  xmlns:Z="http://www.w3.com/standards/z39.50">
  <D:response>
    <D:href>http://www.foo.com/bar.html</D:href>
    <D:propstat>
      <D:prop><Z:Author/></D:prop>
      <D:status>HTTP/1.1 424 Failed Dependency</D:status>
    </D:propstat>
    <D:propstat>
      <D:prop><Z:Copyright-Owner/></D:prop>
      <D:status>HTTP/1.1 409 Conflict</D:status>
    </D:propstat>
    <D:responsedescription> Copyright Owner can not be deleted
  </D:response>
</D:multistatus>
```

In this example, the client requests the server to set the value of the `http://www.w3.com/standards/z39.50/Authors` property, and to remove the property `http://www.w3.com/standards/z39.50/Copyright-Owner`. Since the Copyright-Owner property could not be removed, no property modifications occur. The 424 (Failed Dependency) status code for the Authors property indicates this action would have succeeded if it were not for the conflict with removing the Copyright-Owner property.

8.3 MKCOL Method

The MKCOL method is used to create a new collection. All DAV compliant resources MUST support the MKCOL method.

8.3.1 Request

MKCOL creates a new collection resource at the location specified by the Request-URI. If the resource identified by the Request-URI is non-null then the MKCOL MUST fail. During MKCOL processing, a server MUST make the Request-URI a member of its parent collection, unless the Request-URI is ".". If no such ancestor exists, the method MUST fail. When the MKCOL operation creates a new collection resource, all ancestors MUST already exist, or the method MUST fail with a 409 (Conflict) status code. For example, if a request to create collection `/a/b/c/d/` is made, and neither `/a/b/` nor `/a/b/c/` exists, the request must fail.

When MKCOL is invoked without a request body, the newly created collection SHOULD have no members.

A MKCOL request message may contain a message body. The behavior of a MKCOL request when the body is present is limited to creating collections, members of a collection, bodies of members and properties on the collections or members. If the server receives a MKCOL request entity type it does not support or understand it MUST respond with a 415 (Unsupported Media Type) status code. The exact behavior of MKCOL for various request media types is undefined in this document, and will be specified in separate documents.

8.3.2 Status Codes

Responses from a MKCOL request MUST NOT be cached as MKCOL has non-idempotent semantics.

201 (Created) - The collection or structured resource was created in its entirety.

403 (Forbidden) - This indicates at least one of two conditions: 1) the server does not allow the creation of collections at the given location in its namespace, or 2) the parent collection of the Request-URI exists but cannot accept members.

405 (Method Not Allowed) - MKCOL can only be executed on a deleted/non-existent resource.

409 (Conflict) - A collection cannot be made at the Request-URI until one or more intermediate collections have been created.

415 (Unsupported Media Type) - The server does not support the request type of the body.

507 (Insufficient Storage) - The resource does not have sufficient space to record the state of the resource after the execution of this method.

8.3.3 Example - MKCOL

This example creates a collection called `/webdisc/xfiles/` on the server `www.server.org`.

>>Request

```
MKCOL /webdisc/xfiles/ HTTP/1.1
Host: www.server.org
```

>>Response

```
HTTP/1.1 201 Created
```

8.4 GET, HEAD for Collections

The semantics of GET are unchanged when applied to a collection, since GET is defined as, "retrieve whatever information (in the form of an entity) is identified by the Request-URI" [RFC2068]. GET when applied to a collection may return the contents of an "index.html" resource, a human-readable view of the contents of the collection, or something else altogether. Hence it is possible that the result of a GET on a collection will bear no correlation to the membership of the collection.

Similarly, since the definition of HEAD is a GET without a response message body, the semantics of HEAD are unmodified when applied to collection resources.

8.5 POST for Collections

Since by definition the actual function performed by POST is determined by the server and often depends on the particular resource, the behavior of POST when applied to collections cannot be meaningfully modified because it is largely undefined. Thus the semantics of POST are unmodified when applied to a collection.

8.6 DELETE

8.6.1 DELETE for Non-Collection Resources

If the DELETE method is issued to a non-collection resource whose URIs are an internal member of one or more collections, then during DELETE processing a server MUST remove any URI for the resource identified by the Request-URI from collections which contain it as a member.

8.6.2 DELETE for Collections

The DELETE method on a collection MUST act as if a "Depth: infinity" header was used on it. A client MUST NOT submit a Depth header with a DELETE on a collection with any value but infinity.

DELETE instructs that the collection specified in the Request-URI and all resources identified by its internal member URIs are to be deleted.

If any resource identified by a member URI cannot be deleted then all of the member's ancestors MUST NOT be deleted, so as to maintain namespace consistency.

Any headers included with DELETE MUST be applied in processing every resource to be deleted.

When the DELETE method has completed processing it MUST result in a consistent namespace.

If an error occurs with a resource other than the resource identified in the Request-URI then the response MUST be a 207 (Multi-Status). 424 (Failed Dependency) errors SHOULD NOT be in the 207 (Multi-Status). They can be safely left out because the client will know that the ancestors of a resource could not be deleted when the client receives an error for the ancestor's progeny. Additionally 204 (No Content) errors SHOULD NOT be returned in the 207 (Multi-Status). The reason for this prohibition is that 204 (No Content) is the default success code.

8.6.2.1 Example - DELETE

>>Request

```
DELETE /container/ HTTP/1.1
Host: www.foo.bar
```

>>Response

```
HTTP/1.1 207 Multi-Status
Content-Type: text/xml; charset=utf-8
Content-Length: xxxx

<?xml version="1.0" encoding="utf-8" ?>
<d:multistatus xmlns:d="DAV:">
  <d:response>
    <d:href>http://www.foo.bar/container/resource3</d:href>
    <d:status>HTTP/1.1 423 Locked</d:status>
  </d:response>
</d:multistatus>
```

In this example the attempt to delete `http://www.foo.bar/container/resource3` failed because it is locked, and no lock token was submitted with the request. Consequently, the attempt to delete `http://www.foo.bar/container/` also failed. Thus the client knows that the attempt to delete `http://www.foo.bar/container/` must have also failed since the parent can not be deleted unless its child has also been deleted. Even though a Depth header has not been included, a depth of infinity is assumed because the method is on a collection.

8.7 PUT

8.7.1 PUT for Non-Collection Resources

A PUT performed on an existing resource replaces the GET response entity of the resource. Properties defined on the resource may be recomputed during PUT processing but are not otherwise affected. For example, if a server recognizes the content type of the request body, it may be able to automatically extract information that could be profitably exposed as properties.

A PUT that would result in the creation of a resource without an appropriately scoped parent collection MUST fail with a 409 (Conflict).

8.7.2 PUT for Collections

As defined in the HTTP/1.1 specification [RFC2068], the "PUT" method requests that the enclosed entity be stored under the supplied Request-URI." Since submission of an entity representing a collection would implicitly encode creation and deletion of resources, this specification intentionally does not define a transmission format for creating a collection using PUT. Instead, the MKCOL method is defined to create collections.

When the PUT operation creates a new non-collection resource all ancestors MUST already exist. If all ancestors do not exist, the method MUST fail with a 409 (Conflict) status code. For example, if resource `/a/b/c/d.html` is to be created and `/a/b/c/` does not exist, then the request must fail.

8.8 COPY Method

The COPY method creates a duplicate of the source resource, identified by the Request-URI, in the destination resource, identified by the URI in the Destination header. The Destination header MUST be present. The exact behavior of the COPY method depends on the type of the source resource.

All WebDAV compliant resources MUST support the COPY method. However, support for the COPY method does not guarantee the ability to copy a resource. For example, separate programs may control resources on the same server. As a result, it may not be possible to copy a resource to a location that appears to be on the same server.

8.8.1 COPY for HTTP/1.1 resources

When the source resource is not a collection the result of the COPY method is the creation of a new resource at the destination whose state and behavior match that of the source resource as closely as possible. After a successful COPY invocation, all properties on the source resource MUST be duplicated on the destination resource, subject to modifying headers and XML elements, following the definition for copying properties. Since the environment at the destination may be different than at the source due to factors outside the scope of control of the server, such as the absence of resources required for correct operation, it may not be possible to completely duplicate the behavior of the resource at the destination. Subsequent alterations to the destination resource will not modify the source resource. Subsequent alterations to the source resource will not modify the destination resource.

8.8.2 COPY for Properties

The following section defines how properties on a resource are handled during a COPY operation.

Live properties SHOULD be duplicated as identically behaving live properties at the destination resource. If a property cannot be copied live, then its value MUST be duplicated, octet-for-octet, in an identically named, dead property on the destination resource subject to the effects of the propertybehavior XML element.

The propertybehavior XML element can specify that properties are copied on best effort, that all live properties must be successfully copied or the method must fail, or that a specified list of live properties must be successfully copied or the method must fail. The propertybehavior XML element is defined in section 12.12.

8.8.3 COPY for Collections

The COPY method on a collection without a Depth header MUST act as if a Depth header with value "infinity" was included. A client may submit a Depth header on a COPY on a collection with a value of "0" or "infinity". DAV compliant servers MUST support the "0" and "infinity" Depth header behaviors.

A COPY of depth infinity instructs that the collection resource identified by the Request-URI is to be copied to the location identified by the URI in the Destination header, and all its internal member resources are to be copied to a location relative to it, recursively through all levels of the collection hierarchy.

A COPY of "Depth: 0" only instructs that the collection and its properties but not resources identified by its internal member URIs, are to be copied.

Any headers included with a COPY MUST be applied in processing every resource to be copied with the exception of the Destination header.

The Destination header only specifies the destination URI for the Request-URI. When applied to members of the collection identified by the Request-URI the value of Destination is to be modified

to reflect the current location in the hierarchy. So, if the Request-URI is /a/ with Host header value http://fun.com/ and the Destination is http://fun.com/b/ then when http://fun.com/a/c/d is processed it must use a Destination of http://fun.com/b/c/d.

When the COPY method has completed processing it MUST have created a consistent namespace at the destination (see section 5.1 for the definition of namespace consistency). However, if an error occurs while copying an internal collection, the server MUST NOT copy any resources identified by members of this collection (i.e., the server must skip this subtree), as this would create an inconsistent namespace. After detecting an error, the COPY operation SHOULD try to finish as much of the original copy operation as possible (i.e., the server should still attempt to copy other subtrees and their members, that are not descendants of an error-causing collection). So, for example, if an infinite depth copy operation is performed on collection /a/, which contains collections /a/b/ and /a/c/, and an error occurs copying /a/b/, an attempt should still be made to copy /a/c/. Similarly, after encountering an error copying a non-collection resource as part of an infinite depth copy, the server SHOULD try to finish as much of the original copy operation as possible.

If an error in executing the COPY method occurs with a resource other than the resource identified in the Request-URI then the response MUST be a 207 (Multi-Status).

The 424 (Failed Dependency) status code SHOULD NOT be returned in the 207 (Multi-Status) response from a COPY method. These responses can be safely omitted because the client will know that the progeny of a resource could not be copied when the client receives an error for the parent. Additionally 201 (Created)/204 (No Content) status codes SHOULD NOT be returned as values in 207 (Multi-Status) responses from COPY methods. They, too, can be safely omitted because they are the default success codes.

8.8.4 COPY and the Overwrite Header

If a resource exists at the destination and the Overwrite header is "T" then prior to performing the copy the server MUST perform a DELETE with "Depth: infinity" on the destination resource. If the Overwrite header is set to "F" then the operation will fail.

8.8.5 Status Codes

201 (Created) - The source resource was successfully copied. The copy operation resulted in the creation of a new resource.

204 (No Content) - The source resource was successfully copied to a pre-existing destination resource.

403 (Forbidden) - The source and destination URIs are the same.

409 (Conflict) - A resource cannot be created at the destination until one or more intermediate collections have been created.

412 (Precondition Failed) - The server was unable to maintain the liveness of the properties listed in the propertybehavior XML element or the Overwrite header is "F" and the state of the destination resource is non-null.

423 (Locked) - The destination resource was locked.

502 (Bad Gateway) - This may occur when the destination is on another server and the destination server refuses to accept the resource.

507 (Insufficient Storage) - The destination resource does not have sufficient space to record the state of the resource after the execution of this method.

8.8.6 Example - COPY with Overwrite

This example shows resource `http://www.ics.uci.edu/~fielding/index.html` being copied to the location `http://www.ics.uci.edu/users/fielding/index.html`. The 204 (No Content) status code indicates the existing resource at the destination was overwritten.

>>Request

```
COPY /~fielding/index.html HTTP/1.1
Host: www.ics.uci.edu
Destination: http://www.ics.uci.edu/users/fielding/index.html
```

>>>Response

```
HTTP/1.1 204 No Content
```

8.8.7 Example - COPY with No Overwrite

The following example shows the same copy operation being performed, but with the Overwrite header set to "F". A response of 412 (Precondition Failed) is returned because the destination resource has a non-null state.

>>Request

```
COPY /~fielding/index.html HTTP/1.1
Host: www.ics.uci.edu
Destination: http://www.ics.uci.edu/users/fielding/index.html
Overwrite: F
```

>>>Response

```
HTTP/1.1 412 Precondition Failed
```

8.8.8 Example - COPY of a Collection

>>Request

```
COPY /container/ HTTP/1.1
Host: www.foo.bar
Destination: http://www.foo.bar/othercontainer/
Depth: infinity
Content-Type: text/xml; charset=utf-8
Content-Length: xxxx
```

```
<?xml version="1.0" encoding="utf-8" ?>
<d:propertybehavior xmlns:d="DAV:">
  <d:keepalive></d:keepalive>
</d:propertybehavior>
```

>>>Response

```
HTTP/1.1 207 Multi-Status
Content-Type: text/xml; Charset=utf-8
Content-Length: xxxx

<?xml version="1.0" encoding="utf-8" ?>
<d:multistatus xmlns:d="DAV:">
  <d:response>
    <d:href>http://www.foo.bar/othercontainer/R2/</d:href>
    <d:status>HTTP/1.1 412 Precondition Failed</d:status>
  </d:response>
</d:multistatus>
```

The Depth header is unnecessary as the default behavior of COPY on a collection is to act as if a "Depth: infinity" header had been submitted. In this example most of the resources, along with the collection, were copied successfully. However the collection R2 failed, most likely due to a problem with maintaining the liveness of properties (this is specified by the propertybehavior XML element). Because there was an error copying R2, none of R2's members were copied. However no errors were listed for those members due to the error minimization rules given in section 8.8.3.

8.9 MOVE Method

The MOVE operation on a non-collection resource is the logical equivalent of a copy (COPY), followed by consistency maintenance processing, followed by a delete of the source, where all three actions are performed atomically. The consistency maintenance step allows the server to perform updates caused by the move, such as updating all URIs other than the Request-URI which identify the source resource, to point to the new destination resource. Consequently, the Destination header MUST be present on all MOVE methods and MUST follow all COPY requirements for the COPY part of the MOVE method. All DAV compliant resources MUST support the MOVE method. However, support for the MOVE method does not guarantee the ability to move a resource to a particular destination.

For example, separate programs may actually control different sets of resources on the same server. Therefore, it may not be possible to move a resource within a namespace that appears to belong to the same server.

If a resource exists at the destination, the destination resource will be DELETED as a side-effect of the MOVE operation, subject to the restrictions of the Overwrite header.

8.9.1 MOVE for Properties

The behavior of properties on a MOVE, including the effects of the propertybehavior XML element, MUST be the same as specified in section 8.8.2.

8.9.2 MOVE for Collections

A MOVE with "Depth: infinity" instructs that the collection identified by the Request-URI be moved to the URI specified in the Destination header, and all resources identified by its internal member URIs are to be moved to locations relative to it, recursively through all levels of the collection hierarchy.

The MOVE method on a collection MUST act as if a "Depth: infinity" header was used on it. A client MUST NOT submit a Depth header on a MOVE on a collection with any value but "infinity".

Any headers included with MOVE MUST be applied in processing every resource to be moved with the exception of the Destination header.

The behavior of the Destination header is the same as given for COPY on collections.

When the MOVE method has completed processing it MUST have created a consistent namespace at both the source and destination (see section 5.1 for the definition of namespace consistency). However, if an error occurs while moving an internal collection, the server MUST NOT move any resources identified by members of the failed collection (i.e., the server must skip the error-causing subtree), as this would create an inconsistent namespace. In this case, after detecting the error, the move operation SHOULD try to finish as much of the original move as possible (i.e., the server should still attempt to move other subtrees and the resources identified by their members, that are not descendants of an error-causing collection). So, for example, if an infinite depth move is performed on collection /a/, which contains collections /a/b/ and /a/c/, and an error occurs moving /a/b/, an attempt should still be made to try moving /a/c/. Similarly, after encountering an error moving a non-collection resource as part of an infinite depth move, the server SHOULD try to finish as much of the original move operation as possible.

If an error occurs with a resource other than the resource identified in the Request-URI then the response MUST be a 207 (Multi-Status).

The 424 (Failed Dependency) status code SHOULD NOT be returned in the 207 (Multi-Status) response from a MOVE method. These errors can be safely omitted because the client will know that the progeny of a resource could not be moved when the client receives an error for the parent. Additionally 201 (Created)/204 (No Content) responses SHOULD NOT be returned as values in 207 (Multi-Status) responses from a MOVE. These responses can be safely omitted because they are the default success codes.

8.9.3 MOVE and the Overwrite Header

If a resource exists at the destination and the Overwrite header is "T" then prior to performing the move the server MUST perform a DELETE with "Depth: infinity" on the destination resource. If the Overwrite header is set to "F" then the operation will fail.

8.9.4 Status Codes

201 (Created) - The source resource was successfully moved, and a new resource was created at the destination.

204 (No Content) - The source resource was successfully moved to a pre-existing destination resource.

403 (Forbidden) - The source and destination URIs are the same.

409 (Conflict) - A resource cannot be created at the destination until one or more intermediate collections have been created.

412 (Precondition Failed) - The server was unable to maintain the liveness of the properties listed in the propertybehavior XML element or the Overwrite header is "F" and the state of the destination resource is non-null.

423 (Locked) - The source or the destination resource was locked.

502 (Bad Gateway) - This may occur when the destination is on another server and the destination server refuses to accept the resource.

8.9.5 Example - MOVE of a Non-Collection

This example shows resource `http://www.ics.uci.edu/~fielding/index.html` being moved to the location `http://www.ics.uci.edu/users/fielding/index.html`. The contents of the destination resource would have been overwritten if the destination resource had been non-null. In this case, since there was nothing at the destination resource, the response code is 201 (Created).

>>Request

```
MOVE /~fielding/index.html HTTP/1.1
Host: www.ics.uci.edu
Destination: http://www.ics.uci.edu/users/fielding/index.html
```

>>Response

```
HTTP/1.1 201 Created
Location: http://www.ics.uci.edu/users/fielding/index.html
```

8.9.6 Example - MOVE of a Collection

>>Request

```
MOVE /container/ HTTP/1.1
Host: www.foo.bar
Destination: http://www.foo.bar/othercontainer/
Overwrite: F
If: (<opaquelocktoken:fe184f2e-66ec-41d0-c765-01adc566bb4>)
(<opaquelocktoken:e454f3f3-acdc-452a-56c7-00a5c91e4b77>)
Content-Type: text/xml; charset=utf-8*
Content-Length: xxxxx
```

```
<?xml version="1.0" encoding="utf-8" ?>
<d:propertybehavior xmlns:d="DAV:">
  <d:keepalive></d:keepalive>
</d:propertybehavior>
```

>>Response

```
HTTP/1.1 207 Multi-Status
Content-Type: text/xml; charset=utf-8*
Content-Length: xxxxx
```

```
<?xml version="1.0" encoding="utf-8" ?>
<d:multistatus xmlns:d="DAV:">
```

```
<d:response>
  <d:href>http://www.foo.bar/othercontainer/C2/</d:href>
  <d:status>HTTP/1.1 423 Locked</d:status>
</d:response>
</d:multistatus>
```

In this example the client has submitted a number of lock tokens with the request. A lock token will need to be submitted for every resource, both source and destination, anywhere in the scope of the method, that is locked. In this case the proper lock token was not submitted for the destination `http://www.foo.bar/othercontainer/C2/`. This means that the resource `/container/C2/` could not be moved. Because there was an error copying `/container/C2/`, none of `/container/C2/`'s members were copied. However no errors were listed for those members due to the error minimization rules given in section 8.8.3. User agent authentication has previously occurred via a mechanism outside the scope of the HTTP protocol, in an underlying transport layer.

8.10 LOCK Method

The following sections describe the LOCK method, which is used to take out a lock of any access type. These sections on the LOCK method describe only those semantics that are specific to the LOCK method and are independent of the access type of the lock being requested.

Any resource which supports the LOCK method MUST, at minimum, support the XML request and response formats defined herein.

8.10.1 Operation

A LOCK method invocation creates the lock specified by the lockinfo XML element on the Request-URI. Lock method requests SHOULD have a XML request body which contains an owner XML element for this lock request, unless this is a refresh request. The LOCK request may have a Timeout header.

Clients MUST assume that locks may arbitrarily disappear at any time, regardless of the value given in the Timeout header. The Timeout header only indicates the behavior of the server if "extraordinary" circumstances do not occur. For example, an administrator may remove a lock at any time or the system may crash in such a way that it loses the record of the lock's existence. The response MUST contain the value of the lockdiscovery property in a prop XML element.

The current lock state of a resource is given in the leftmost column, and lock requests are listed in the first row. The intersection of a row and column gives the result of a lock request. For example, if a shared lock is held on a resource, and an exclusive lock is requested, the table entry is "false", indicating the lock must not be granted.

8.10.7 Status Codes

- 200 (OK) - The lock request succeeded and the value of the lockdiscovery property is included in the body.
- 412 (Precondition Failed) - The included lock token was not enforceable on this resource or the server could not satisfy the request in the lockinfo XML element.
- 423 (Locked) - The resource is locked, so the method has been rejected.

8.10.8 Example - Simple Lock Request

>>Request

```
LOCK /workspace/webdav/proposal.doc HTTP/1.1
Host: webdav.sb.aol.com
Timeout: Infinite, Second=4100000000
Content-Type: text/xml; charset=utf-8
Content-Length: xxxx
Authorization: Digest username="ejw",
realm="ejw@webdav.sb.aol.com", nonce="...",
response="...", opaque="..."

<?xml version="1.0" encoding="utf-8" ?>
<D:lockinfo xmlns:D="DAV:">
  <D:lockscope><D:exclusive/></D:lockscope>
  <D:locktype><D:write/></D:locktype>
  <D:owner>
    <D:href>http://www.ics.uci.edu/~ejw/contact.html</D:href>
  </D:owner>
</D:lockinfo>
```

>>Response

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: xxxx

<?xml version="1.0" encoding="utf-8" ?>
<D:prop xmlns:D="DAV:">
  <D:lockdiscovery>
    <D:activelock>
      <D:locktype><D:write/></D:locktype>
      <D:lockscope><D:exclusive/></D:lockscope>
      <D:depth>Infinity</D:depth>
      <D:owner>
        <D:href>
          http://www.ics.uci.edu/~ejw/contact.html
        </D:href>
      </D:owner>
      <D:timeout>Second=604800</D:timeout>
      <D:locktoken>
        <D:href>
          opaque:locktoken:e71d4fae-5dec-22d6-fea5-00a0c91e6be4
        </D:href>
      </D:locktoken>
    </D:activelock>
    </D:lockdiscovery>
  </D:prop>
```

In order to indicate the lock token associated with a newly created lock, a Lock-Token response header MUST be included in the response for every successful LOCK request for a new lock. Note that the Lock-Token header would not be returned in the response for a successful refresh LOCK request because a new lock was not created.

8.10.2 The Effect of Locks on Properties and Collections

The scope of a lock is the entire state of the resource, including its body and associated properties. As a result, a lock on a resource MUST also lock the resource's properties.

For collections, a lock also affects the ability to add or remove members. The nature of the effect depends upon the type of access control involved.

8.10.3 Locking Replicated Resources

A resource may be made available through more than one URI. However locks apply to resources, not URIs. Therefore a LOCK request on a resource MUST NOT succeed if it can not be honored by all the URIs through which the resource is addressable.

8.10.4 Depth and Locking

The Depth header may be used with the LOCK method. Values other than 0 or infinity MUST NOT be used with the Depth header on a LOCK method. All resources that support the LOCK method MUST support the Depth header.

A Depth header of value 0 means to just lock the resource specified by the Request-URI.

If the Depth header is set to infinity then the resource specified in the Request-URI along with all its internal members, all the way down the hierarchy, are to be locked. A successful result MUST return a single lock token which represents all the resources that have been locked. If an UNLOCK is successfully executed on this token, all associated resources are unlocked. If the lock cannot be granted to all resources, a 409 (Conflict) status code MUST be returned with a response entity body containing a multistatus XML element describing which resource(s) prevented the lock from being granted. Hence, partial success is not an option. Either the entire hierarchy is locked or no resources are locked.

If no Depth header is submitted on a LOCK request then the request MUST act as if a "Depth:infinity" had been submitted.

8.10.5 Interaction with other Methods

The interaction of a LOCK with various methods is dependent upon the lock type. However, independent of lock type, a successful DELETE of a resource MUST cause all of its locks to be removed.

8.10.6 Lock Compatibility Table

The table below describes the behavior that occurs when a lock request is made on a resource.

CURRENT LOCK STATE/ LOCK REQUEST	SHARED LOCK	EXCLUSIVE LOCK
None	True	True
Shared Lock	True	False
Exclusive Lock	False	False*

Legend: True = lock may be granted. False = lock MUST NOT be granted. *It is illegal for a principal to request the same lock twice.

8.10.10 Example - Multi-Resource Lock Request

>>>Request

```
LOCK /webdav/ HTTP/1.1
Host: webdav.sb.aol.com
Timeout: Infinite, Second=4100000000
Depth: infinity
Content-Type: text/xml; charset=utf-8
Content-Length: xxxx
Authorization: Digest username="ejw",
    realm="ejw@webdav.sb.aol.com", nonce="...",
    uri="/workspace/webdav/proposal.doc",
    response="...", opaque="..."
<?xml version="1.0" encoding="utf-8" ?>
<D:lockinfo xmlns:D="DAV:">
  <D:locktype><D:write/></D:locktype>
  <D:lockscope><D:exclusive/></D:lockscope>
  <D:owner>
    <D:href>http://www.ics.uci.edu/~ejw/contact.html</D:href>
  </D:owner>
</D:lockinfo>
```

>>>Response

```
HTTP/1.1 207 Multi-Status
Content-Type: text/xml; charset=utf-8
Content-Length: xxxx
<?xml version="1.0" encoding="utf-8" ?>
<D:multistatus xmlns:D="DAV:">
  <D:response>
    <D:href>http://webdav.sb.aol.com/webdav/secret</D:href>
    <D:status>HTTP/1.1 403 Forbidden</D:status>
  </D:response>
  <D:response>
    <D:href>http://webdav.sb.aol.com/webdav/</D:href>
    <D:propstat>
      <D:prop><D:lockdiscovery/></D:prop>
      <D:status>HTTP/1.1 424 Failed Dependency</D:status>
    </D:propstat>
  </D:response>
</D:multistatus>
```

This example shows a request for an exclusive write lock on a collection and all its children. In this request, the client has specified that it desires an infinite length lock, if available, otherwise a timeout of 4.1 billion seconds, if available. The request entity body contains the contact information for the principal taking out the lock, in this case a web page URL.

The error is a 403 (Forbidden) response on the resource <http://webdav.sb.aol.com/webdav/secret>. Because this resource could not be locked, none of the resources were locked. Note also that the lockdiscovery property for the Request-URI has been included as required. In this example the lockdiscovery property is empty which means that there are no outstanding locks on the resource.

In this example, the nonce, response, and opaque fields have not been calculated in the Authorization request header.

This example shows the successful creation of an exclusive write lock on resource <http://webdav.sb.aol.com/workspace/webdav/proposal.doc>. The resource <http://www.ics.uci.edu/~ejw/contact.html> contains contact information for the owner of the lock. The server has an activity-based timeout policy in place on this resource, which causes the lock to automatically be removed after 1 week (604800 seconds). Note that the nonce, response, and opaque fields have not been calculated in the Authorization request header.

8.10.9 Example - Refreshing a Write Lock

>>>Request

```
LOCK /workspace/webdav/proposal.doc HTTP/1.1
Host: webdav.sb.aol.com
Timeout: Infinite, Second=4100000000
If: <opaque:locktoken:e71d4fae-5dec-22d6-fea5-00a0c91a6be4>
Authorization: Digest username="ejw",
    realm="ejw@webdav.sb.aol.com", nonce="...",
    uri="/workspace/webdav/proposal.doc",
    response="...", opaque="..."
```

>>>Response

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: xxxx
<?xml version="1.0" encoding="utf-8" ?>
<D:prop xmlns:D="DAV:">
  <D:lockdiscovery>
    <D:activelock>
      <D:locktype><D:write/></D:locktype>
      <D:lockscope><D:exclusive/></D:lockscope>
      <D:depth>infinity</D:depth>
      <D:owner>
        <D:href>
          http://www.ics.uci.edu/~ejw/contact.html
        </D:href>
      </D:owner>
      <D:timeout>Second=604800</D:timeout>
      <D:locktoken>
        <D:href>
          opaque:locktoken:e71d4fae-5dec-22d6-fea5-00a0c91a6be4
        </D:href>
      </D:locktoken>
    </D:activelock>
  </D:lockdiscovery>
</D:prop>
```

This request would refresh the lock, resetting any time outs. Notice that the client asked for an infinite time out but the server choose to ignore the request. In this example, the nonce, response, and opaque fields have not been calculated in the Authorization request header.

8.11 UNLOCK Method

The UNLOCK method removes the lock identified by the lock token in the Lock-Token request header from the Request-URI, and all other resources included in the lock. If all resources which have been locked under the submitted lock token can not be unlocked then the UNLOCK request MUST fail.

Any DAV compliant resource which supports the LOCK method MUST support the UNLOCK method.

8.11.1 Example - UNLOCK

>>>Request

```
UNLOCK /workspace/webdav/info.doc HTTP/1.1
Host: webdav.sb.aol.com
Lock-Token: <opaque-locktoken:a515cfa4-5da4-22e1-f5b5-00a0451e6bf7>
Authorization: Digest username="ejw",
  realm="ejw@webdav.sb.aol.com", nonce="...",
  uri="/workspace/webdav/proposal.doc",
  responses="...", opaque="..."
```

>>>Response

HTTP/1.1 204 No Content

In this example, the lock identified by the lock token "opaque-locktoken:a515cfa4-5da4-22e1-f5b5-00a0451e6bf7" is successfully removed from the resource <http://webdav.sb.aol.com/workspace/webdav/info.doc>. If this lock included more than just one resource, the lock is removed from all resources included in the lock. The 204 (No Content) status code is used instead of 200 (OK) because there is no response entity body.

In this example, the nonce, response, and opaque fields have not been calculated in the Authorization request header.

9 HTTP Headers for Distributed Authoring

9.1 DAV Header

DAV = "DAV" ":" "1" ["1" ":" "2"] ["1" ":" "1#extend"]

This header indicates that the resource supports the DAV schema and protocol as specified. All DAV compliant resources MUST return the DAV header on all OPTIONS responses.

The value is a list of all compliance classes that the resource supports. Note that above a comma has already been added to the 2. This is because a resource can not be level 2 compliant unless it is also level 1 compliant. Please refer to section 15 for more details. In general, however, support for one compliance class does not entail support for any other.

9.2 Depth Header

Depth = "Depth" ":" ("0" | "1" | "infinity")

The Depth header is used with methods executed on resources which could potentially have internal members to indicate whether the method is to be applied only to the resource ("Depth: 0"), to the resource and its immediate children, ("Depth: 1"), or the resource and all its progeny ("Depth: infinity").

The Depth header is only supported if a method's definition explicitly provides for such support.

The following rules are the default behavior for any method that supports the Depth header. A method may override these defaults by defining different behavior in its definition.

Methods which support the Depth header may choose not to support all of the header's values and may define, on a case by case basis, the behavior of the method if a Depth header is not present. For example, the MOVE method only supports "Depth: infinity" and if a Depth header is not present will act as if a "Depth: infinity" header had been applied.

Clients MUST NOT rely upon methods executing on members of their hierarchies in any particular order or on the execution being atomic unless the particular method explicitly provides such guarantees.

Upon execution, a method with a Depth header will perform as much of its assigned task as possible and then return a response specifying what it was able to accomplish and what it failed to do.

So, for example, an attempt to COPY a hierarchy may result in some of the members being copied and some not.

Any headers on a method that has a defined interaction with the Depth header MUST be applied to all resources in the scope of the method except where alternative behavior is explicitly defined. For example, an If-Match header will have its value applied against every resource in the method's scope and will cause the method to fail if the header fails to match.

If a resource, source or destination, within the scope of the method with a Depth header is locked in such a way as to prevent the successful execution of the method, then the lock token for that resource MUST be submitted with the request in the If request header.

The Depth header only specifies the behavior of the method with regards to internal children. If a resource does not have internal children then the Depth header MUST be ignored.

Please note, however, that it is always an error to submit a value for the Depth header that is not allowed by the method's definition. Thus submitting a "Depth: 1" on a COPY, even if the resource

does not have internal members, will result in a 400 (Bad Request). The method should fail not because the resource doesn't have internal members, but because of the illegal value in the header.

9.3 Destination Header

Destination = "Destination" ":" absoluteURI

The Destination header specifies the URI which identifies a destination resource for methods such as COPY and MOVE, which take two URIs as parameters. Note that the absoluteURI production is defined in [RFC2396].

9.4 If Header

```
If = "If" ":" ( 1*No-tag-list | 1*Tagged-list )
No-tag-list = List
Tagged-list = Resource 1*List
Resource = Coded-URL
List = "(" 1* ("Not" | (State-token | "[" entity-tag "]" )) ")"
State-token = Coded-URL
Coded-URL = "<" absoluteURI ">"
```

The If header is intended to have similar functionality to the If-Match header defined in section 14.25 of [RFC2068]. However the If header is intended for use with any URI which represents state information, referred to as a state token, about a resource as well as ETags. A typical example of a state token is a lock token, and lock tokens are the only state tokens defined in this specification.

All DAV compliant resources MUST honor the If header.

The If header's purpose is to describe a series of state lists. If the state of the resource to which the header is applied does not match any of the specified state lists then the request MUST fail with a 412 (Precondition Failed). If one of the described state lists matches the state of the resource then the request may succeed.

Note that the absoluteURI production is defined in [RFC2396].

9.4.1 No-tag-list Production

The No-tag-list production describes a series of state tokens and ETags. If multiple No-tag-list productions are used then one only needs to match the state of the resource for the method to be allowed to continue.

If a method, due to the presence of a Depth or Destination header, is applied to multiple resources then the No-tag-list production MUST be applied to each resource the method is applied to.

9.4.1.1 Example - No-tag-list If Header

```
If: (<locktoken:a-write-lock-token> ["I am an ETag"]) ("I am another ETag")
```

The previous header would require that any resources within the scope of the method must either be locked with the specified lock token and in the state identified by the "I am an ETag" ETag or in the state identified by the second ETag "I am another ETag". To put the matter more plainly one can think of the previous If header as being in the form (or (<locktoken:a-write-lock-token> ["I am an ETag"]) and ("I am another ETag"))).

9.4.2 Tagged-list Production

The tagged-list production scopes a list production. That is, it specifies that the lists following the resource specification only apply to the specified resource. The scope of the resource production

begins with the list production immediately following the resource production and ends with the next resource production, if any.

When the If header is applied to a particular resource, the Tagged-list productions MUST be searched to determine if any of the listed resources match the operand resource(s) for the current method. If none of the resource productions match the current resource then the header MUST be ignored. If one of the resource productions does match the name of the resource under consideration then the list productions following the resource production MUST be applied to the resource in the manner specified in the previous section.

The same URI MUST NOT appear more than once in a resource production in an If header.

9.4.2.1 Example - Tagged List If header

```
COPY /resource1 HTTP/1.1
Host: www.foo.bar
Destination: http://www.foo.bar/resource2
If: <http://www.foo.bar/resource1> (<locktoken:a-write-lock-token>
["/A weak ETag"]) ("strong ETag")
<http://www.foo.bar/random> ("another strong ETag")
```

In this example `http://www.foo.bar/resource1` is being copied to `http://www.foo.bar/resource2`. When the method is first applied to `http://www.foo.bar/resource1`, resource1 must be in the state specified by "<locktoken:a-write-lock-token> ["/A weak ETag"] ("strong ETag")". That is, it either must be locked with a lock token of "locktoken:a-write-lock-token" and have a weak entity tag "/A weak ETag" or it must have a strong entity tag "strong ETag".

That is the only success condition since the resource `http://www.foo.bar/random` never has the method applied to it (the only other resource listed in the If header) and `http://www.foo.bar/resource2` is not listed in the If header.

9.4.3 not Production

Every state token or ETag is either current, and hence describes the state of a resource, or is not current, and does not describe the state of a resource. The boolean operation of matching a state token or ETag to the current state of a resource thus resolves to a true or false value. The not production is used to reverse that value. The scope of the not production is the state-token or entity-tag immediately following it.

```
If: (Not <locktoken:write1> <locktoken:write2>)
```

When submitted with a request, this If header requires that all operand resources must not be locked with `locktoken:write1` and must be locked with `locktoken:write2`.

9.4.4 Matching Function

When performing If header processing, the definition of a matching state token or entity tag is as follows.

Matching entity tag: Where the entity tag matches an entity tag associated with that resource.

Matching state token: Where there is an exact match between the state token in the If header and any state token on the resource.

9.4.5 If Header and Non-DAV Compliant Proxies

Non-DAV compliant proxies will not honor the If header, since they will not understand the If header, and HTTP requires non-understood headers to be ignored. When communicating with HTTP/1.1 proxies, the "Cache-Control: no-cache" request header MUST be used so as to prevent

the proxy from improperly trying to service the request from its cache. When dealing with HTTP/1.0 proxies the "Pragma: no-cache" request header MUST be used for the same reason.

9.5 Lock-Token Header

Lock-Token = "Lock-Token" ":" Coded-URL

The Lock-Token request header is used with the UNLOCK method to identify the lock to be removed. The lock token in the Lock-Token request header MUST identify a lock that contains the resource identified by Request-URI as a member.

The Lock-Token response header is used with the LOCK method to indicate the lock token created as a result of a successful LOCK request to create a new lock.

9.6 Overwrite Header

Overwrite = "Overwrite" ":" ("T" | "F")

The Overwrite header specifies whether the server should overwrite the state of a non-null destination resource during a COPY or MOVE. A value of "T" states that the server must not perform the COPY or MOVE operation if the state of the destination resource is non-null. If the overwrite header is not included in a COPY or MOVE request then the resource MUST treat the request as if it has an overwrite header of value "T". While the Overwrite header appears to duplicate the functionality of the If-Match: * header of HTTP/1.1, If-Match applies only to the Request-URI, and not to the Destination of a COPY or MOVE.

If a COPY or MOVE is not performed due to the value of the Overwrite header, the method MUST fail with a 412 (Precondition Failed) status code.

All DAV compliant resources MUST support the Overwrite header.

9.7 Status-URI Response Header

The Status-URI response header may be used with the 102 (Processing) status code to inform the client as to the status of a method.

Status-URI = "Status-URI" ":" *(Status-Code Coded-URL) ; Status-Code is defined in 6.1.1 of [RFC2068]

The URIs listed in the header are source resources which have been affected by the outstanding method. The status code indicates the resolution of the method on the identified resource. So, for example, if a MOVE method on a collection is outstanding and a 102 (Processing) response with a Status-URI response header is returned, the included URIs will indicate resources that have had move attempted on them and what the result was.

9.8 Timeout Request Header

Timeout = "Timeout" ":" 1#TimeType
TimeType = ("Second-" DAVTimeOutVal | "Infinite" | Other)
DAVTimeOutVal = 1-digit
Other = "Extend" field-value ; See section 4.2 of [RFC2068]

Clients may include Timeout headers in their LOCK requests. However, the server is not required to honor or even consider these requests. Clients MUST NOT submit a Timeout request header with any method other than a LOCK method.

A Timeout request header MUST contain at least one TimeType and may contain multiple TimeType entries. The purpose of listing multiple TimeType entries is to indicate multiple different values and value types that are acceptable to the client. The client lists the TimeType entries in order of preference.

Timeout response values MUST use a Second value, Infinite, or a TimeType the client has indicated familiarity with. The server may assume a client is familiar with any TimeType submitted in a Timeout header.

The "Second" TimeType specifies the number of seconds that will elapse between granting of the lock at the server, and the automatic removal of the lock. The timeout value for TimeType "Second" MUST NOT be greater than 2^32-1.

The timeout counter SHOULD be restarted any time an owner of the lock sends a method to any member of the lock, including unsupported methods, or methods which are unsuccessful. However the lock MUST be refreshed if a refresh LOCK method is successfully received.

If the timeout expires then the lock may be lost. Specifically, if the server wishes to harvest the lock upon time-out, the server SHOULD act as if an UNLOCK method was executed by the server on the resource using the lock token of the timed-out lock, performed with its override authority. Thus logs should be updated with the disposition of the lock, notifications should be sent, etc., just as they would be for an UNLOCK request.

Servers are advised to pay close attention to the values submitted by clients, as they will be indicative of the type of activity the client intends to perform. For example, an applet running in a browser may need to lock a resource, but because of the instability of the environment within which the applet is running, the applet may be turned off without warning. As a result, the applet is likely to ask for a relatively small timeout value so that if the applet dies, the lock can be quickly harvested. However, a document management system is likely to ask for an extremely long timeout because its user may be planning on going off-line.

A client MUST NOT assume that just because the time-out has expired the lock has been lost.

10 Status Code Extensions to HTTP/1.1

The following status codes are added to those defined in HTTP/1.1 [RFC2068].

10.1 102 Processing

The 102 (Processing) status code is an interim response used to inform the client that the server has accepted the complete request, but has not yet completed it. This status code SHOULD only be sent when the server has a reasonable expectation that the request will take significant time to complete. As guidance, if a method is taking longer than 20 seconds (a reasonable, but arbitrary value) to process the server SHOULD return a 102 (Processing) response. The server MUST send a final response after the request has been completed.

Methods can potentially take a long period of time to process, especially methods that support the Depth header. In such cases the client may time-out the connection while waiting for a response. To prevent this the server may return a 102 (Processing) status code to indicate to the client that the server is still processing the method.

10.2 207 Multi-Status

The 207 (Multi-Status) status code provides status for multiple independent operations (see section 11 for more information).

10.3 422 Unprocessable Entity

The 422 (Unprocessable Entity) status code means the server understands the content type of the request entity (hence a 415 (Unsupported Media Type) status code is inappropriate), and the syntax of the request entity is correct (thus a 400 (Bad Request) status code is inappropriate) but was unable to process the contained instructions. For example, this error condition may occur if an XML request body contains well-formed (i.e., syntactically correct), but semantically erroneous XML instructions.

10.4 423 Locked

The 423 (Locked) status code means the source or destination resource of a method is locked.

10.5 424 Failed Dependency

The 424 (Failed Dependency) status code means that the method could not be performed on the resource because the requested action depended on another action and that action failed. For example, if a command in a PROPPATCH method fails then, at minimum, the rest of the commands will also fail with 424 (Failed Dependency).

10.6 507 Insufficient Storage

The 507 (Insufficient Storage) status code means the method could not be performed on the resource because the server is unable to store the representation needed to successfully complete the request. This condition is considered to be temporary. If the request which received this status code was the result of a user action, the request MUST NOT be repeated until it is requested by a separate user action.

11 Multi-Status Response

The default 207 (Multi-Status) response body is a text/xml or application/xml HTTP entity that contains a single XML element called multistatus, which contains a set of XML elements called response which contain 200, 300, 400, and 500 series status codes generated during the method invocation. 100 series status codes SHOULD NOT be recorded in a response XML element.

12 XML Element Definitions

In the section below, the final line of each section gives the element type declaration using the format defined in [REC-XML]. The "Value" field, where present, specifies further restrictions on the allowable contents of the XML element using BNF (i.e., to further restrict the values of a PCDATA element).

12.1 activelock XML Element

Name: activelock
Namespace: DAV:
Purpose: Describes a lock on a resource.
<ELEMENT activelock (lockscope, locktype, depth, owner?, timeout?, locktoken?) >

12.1.1 depth XML Element

Name: depth
Namespace: DAV:
Purpose: The value of the Depth header.
Value: *0 | *1 | *infinity*
<|ELEMENT depth (#PCDATA) >

12.1.2 locktoken XML Element

Name: locktoken
Namespace: DAV:
Purpose: The lock token associated with a lock.
Description: The href contains one or more opaque lock token URIs which all refer to the same lock (i.e., the OpaqueLockToken-URI production in section 6.4).
<|ELEMENT locktoken (href*) >

12.1.3 timeout XML Element

Name: timeout
Namespace: DAV:
Purpose: The timeout associated with a lock
Value: TimeType ; Defined in section 9.8
<|ELEMENT timeout (#PCDATA) >

12.2 collection XML Element

Name: collection
Namespace: DAV:
Purpose: Identifies the associated resource as a collection. The resource type property of a collection resource MUST have this value.

<ELEMENT collection EMPTY >

12.3 href XML Element

Name: href
Namespace: DAV:
Purpose: Identifies the content of the element as a URI.
Value: URI ; See section 3.2.1 of [RFC2068]

<ELEMENT href (#PCDATA)>

12.4 link XML Element

Name: link
Namespace: DAV:
Purpose: Identifies the property as a link and contains the source and destination of that link.
Description: The link XML element is used to provide the sources and destinations of a link. The name of the property containing the link XML element provides the type of the link. Link is a multi-valued element, so multiple links may be used together to indicate multiple links with the same type. The values in the href XML elements inside the src and dst XML elements of the link XML element MUST NOT be rejected if they point to resources which do not exist.

<ELEMENT link (src+, dst+) >

12.4.1 dst XML Element

Name: dst
Namespace: DAV:
Purpose: Indicates the destination of a link
Value: URI

<ELEMENT dst (#PCDATA) >

12.4.2 src XML Element

Name: src
Namespace: DAV:
Purpose: Indicates the source of a link.
Value: URI

<ELEMENT src (#PCDATA) >

12.5 lockentry XML Element

Name: lockentry
Namespace: DAV:
Purpose: Defines the types of locks that can be used with the resource.

<ELEMENT lockentry (lockscope, locktype) >

12.6 lockinfo XML Element

Name: lockinfo
Namespace: DAV:
Purpose: The lockinfo XML element is used with a LOCK method to specify the type of lock the client wishes to have created.

<ELEMENT lockinfo (lockscope, locktype, owner?) >

12.7 lockscope XML Element

Name: lockscope
Namespace: DAV:
Purpose: Specifies whether a lock is an exclusive lock, or a shared lock.

<ELEMENT lockscope (exclusive | shared) >

12.7.1 exclusive XML Element

Name: exclusive
Namespace: DAV:
Purpose: Specifies an exclusive lock

<ELEMENT exclusive EMPTY >

12.7.2 shared XML Element

Name: shared
Namespace: DAV:
Purpose: Specifies a shared lock

<ELEMENT shared EMPTY >

12.8 locktype XML Element

Name: locktype
Namespace: DAV:
Purpose: Specifies the access type of a lock. At present, this specification only defines one lock type, the write lock.

<ELEMENT locktype (write) >

12.8.1 write XML Element

Name: write
Namespace: DAV:
Purpose: Specifies a write lock.

<ELEMENT write EMPTY >

12.10 owner XML Element

Name: owner
Namespace: DAV:
Purpose: Provides information about the principal taking out a lock.
Description: The owner XML element provides information sufficient for either directly contacting a principal (such as a telephone number or Email URL), or for discovering the principal (such as the URL of a homepage) who owns a lock.

<ELEMENT owner ANY>

12.11 prop XML element

Name: prop
Namespace: DAV:
Purpose: Contains properties related to a resource.
Description: The prop XML element is a generic container for properties defined on resources. All elements inside a prop XML element MUST define properties related to the resource. No other elements may be used inside of a prop element.

<ELEMENT prop ANY>

12.12 propertybehavior XML element

Name: propertybehavior
Namespace: DAV:
Purpose: Specifies how properties are handled during a COPY or MOVE.
Description: The propertybehavior XML element specifies how properties are handled during a COPY or MOVE. If this XML element is not included in the request body then the server is expected to act as defined by the default property handling behavior of the associated method. All WebDAV compliant resources MUST support the propertybehavior XML element.

<ELEMENT propertybehavior (omit | keepalive) >

12.12.1 keepalive XML element

Name: keepalive
Namespace: DAV:
Purpose: Specifies requirements for the copying/moving of live properties.
Description: If a list of URLs is included as the value of keepalive then the named properties MUST be "live" after they are copied (moved) to the destination resource of a COPY (or MOVE). If the value "" is given for the keepalive XML element, this designates that all live properties on the source resource MUST be live on the destination. If the requirements specified by the keepalive element can not be honored then the method MUST fail with a 412 (Precondition Failed). All DAV compliant resources MUST support the keepalive XML element for use with the COPY and MOVE methods.

Value: ... ; #PCDATA value can only be ...

<ELEMENT keepalive (#PCDATA | href+) >

12.9 multistatus XML Element

Name: multistatus
Namespace: DAV:
Purpose: Contains multiple response messages.
Description: The multistatus XML element is used to provide a general message describing the overarching nature of the response. If this value is available an application may use it instead of presenting the individual response descriptions contained within the responses.

<ELEMENT multistatus (response*, responsedescription?) >

12.9.1 response XML Element

Name: response
Namespace: DAV:
Purpose: Holds a single response describing the effect of a method on resource and/or its properties.
Description: A particular href MUST NOT appear more than once as the child of a response XML element under a multistatus XML element. This requirement is necessary in order to keep processing costs for a response to linear time. Essentially, this prevents having to search in order to group together all the responses by href. There are, however, no requirements regarding ordering based on href values.

<ELEMENT response (href, (href*, status)|(propstat+)), responsedescription? >

12.9.1.1 propstat XML Element

Name: propstat
Namespace: DAV:
Purpose: Groups together a prop and status element that is associated with a particular href element.
Description: The propstat XML element MUST contain one prop XML element and one status XML element. The contents of the prop XML element MUST only list the names of properties to which the result in the status element applies.

<ELEMENT propstat (prop, status, responsedescription?) >

12.9.1.2 status XML Element

Name: status
Namespace: DAV:
Purpose: Holds a single HTTP status-line
Value: status-line ; status-line defined in [RFC2068]

<ELEMENT status (#PCDATA) >

12.9.2 responsedescription XML Element

Name: responsedescription
Namespace: DAV:
Purpose: Contains a message that can be displayed to the user explaining the nature of the response.
Description: This XML element provides information suitable to be presented to a user.

<ELEMENT responsedescription (#PCDATA) >

12.14 propfind XML Element

Name: propfind
 Namespace: DAV:
 Purpose: Specifies the properties to be returned from a PROPFIND method. Two special elements are specified for use with propfind, allprop and proppatch. If prop is used inside propfind it MUST only contain property names, not values.

<ELEMENT propfind (allprop | proppatch | prop) >

12.14.1 allprop XML Element

Name: allprop
 Namespace: DAV:
 Purpose: The allprop XML element specifies that all property names and values on the resource are to be returned.

<ELEMENT allprop EMPTY >

12.14.2 proppatch XML Element

Name: proppatch
 Namespace: DAV:
 Purpose: The proppatch XML element specifies that only a list of property names on the resource is to be returned.

<ELEMENT proppatch EMPTY >

12.12.2 omit XML element

Name: omit
 Namespace: DAV:
 Purpose: The omit XML element instructs the server that it should use best effort to copy properties but a failure to copy a property MUST NOT cause the method to fail.
 Description: The default behavior for a COPY or MOVE is to copy/move all properties or fail the method. In certain circumstances, such as when a server copies a resource over another protocol such as FTP, it may not be possible to copy/move the properties associated with the resource. This any attempt to copy/move over FTP would always have to fail because properties could not be moved over, even as dead properties. All DAV compliant resources MUST support the omit XML element on COPY/MOVE methods.

<ELEMENT omit EMPTY >

12.13 propertyupdate XML element

Name: propertyupdate
 Namespace: DAV:
 Purpose: Contains a request to alter the properties on a resource.
 Description: This XML element is a container for the information required to modify the properties on the resource. This XML element is multi-valued.

<ELEMENT propertyupdate (remove | set)+ >

12.13.1 remove XML element

Name: remove
 Namespace: DAV:
 Purpose: Lists the DAV properties to be removed from a resource.
 Description: Remove instructs that the properties specified in prop should be removed. Specifying the removal of a property that does not exist is not an error. All the XML elements in a prop XML element inside of a remove XML element MUST be empty, as only the names of properties to be removed are required.

<ELEMENT remove (prop) >

12.13.2 set XML element

Name: set
 Namespace: DAV:
 Purpose: Lists the DAV property values to be set for a resource.
 Description: The set XML element MUST contain only a prop XML element. The elements contained by the prop XML element inside the set XML element MUST specify the name and value of properties that are set on the resource identified by Request-URI. If a property already exists then its value is replaced. Language tagging information in the property's value (in the "xml:lang" attribute, if present) MUST be persistently stored along with the property, and MUST be subsequently retrievable using PROPFIND.

<ELEMENT set (prop) >

13 DAV Properties

For DAV properties, the name of the property is also the same as the name of the XML element that contains its value. In the section below, the final line of each section gives the element type declaration using the format defined in [REC-XML]. The "Value" field, where present, specifies further restrictions on the allowable contents of the XML element using BNF (i.e., to further restrict the values of a PCDATA element).

13.1 creationdate Property

Name: creationdate
Namespace: DAV:
Purpose: Records the time and date the resource was created.
Value: date-time ; See Appendix 2
Description: The creationdate property should be defined on all DAV compliant resources. If present, it contains a timestamp of the moment when the resource was created (i.e., the moment it had non-null state).

<!ELEMENT creationdate (#PCDATA) >

13.2 displayname Property

Name: displayname
Namespace: DAV:
Purpose: Provides a name for the resource that is suitable for presentation to a user.
Description: The displayname property should be defined on all DAV compliant resources. If present, the property contains a description of the resource that is suitable for presentation to a user.

<!ELEMENT displayname (#PCDATA) >

13.3 getcontentlength Property

Name: getcontentlength
Namespace: DAV:
Purpose: Contains the Content-Length header returned by a GET without accept headers
Description: The getcontentlength property MUST be defined on any DAV compliant resource that returns the Content-Language header on a GET.
Value: language-tag ; language-tag is defined in section 14.13 of [RFC2068]

<!ELEMENT getcontentlength (#PCDATA) >

13.4 getcontentlength Property

Name: getcontentlength
Namespace: DAV:
Purpose: Contains the Content-Length header returned by a GET without accept headers.
Description: The getcontentlength property MUST be defined on any DAV compliant resource that returns the Content-Length header in response to a GET.
Value: content-length ; see section 14.14 of [RFC2068]

<!ELEMENT getcontentlength (#PCDATA) >

13.5 getcontenttype Property

Name: getcontenttype
Namespace: DAV:
Purpose: Contains the Content-Type header returned by a GET without accept headers.
Description: This getcontenttype property MUST be defined on any DAV compliant resource that returns the Content-Type header in response to a GET.
Value: media-type ; defined in section 3.7 of [RFC2068]

<!ELEMENT getcontenttype (#PCDATA) >

13.6 getetag Property

Name: getetag
Namespace: DAV:
Purpose: Contains the ETag header returned by a GET without accept headers.
Description: The getetag property MUST be defined on any DAV compliant resource that returns the Etag header.
Value: entity-tag ; defined in section 3.11 of [RFC2068]

<!ELEMENT getetag (#PCDATA) >

13.7 getlastmodified Property

Name: getlastmodified
Namespace: DAV:
Purpose: Contains the Last-Modified header returned by a GET method without accept headers.
Description: Note that the last-modified date on a resource may reflect changes in any part of the state of the resource, not necessarily just a change to the response to the GET method. For example, a change in a property may cause the last-modified date to change. The getlastmodified property MUST be defined on any DAV compliant resource that returns the Last-Modified header in response to a GET.
Value: HTTP-date ; defined in section 3.3.1 of [RFC2068]

<!ELEMENT getlastmodified (#PCDATA) >

13.8 lockdiscovery Property

Name: lockdiscovery
Namespace: DAV:
Purpose: Describes the active locks on a resource
Description: The lockdiscovery property returns a listing of who has a lock, what type of lock he has, the timeout type and the time remaining on the timeout, and the associated lock token. The server is free to withhold any or all of this information if the requesting principal does not have sufficient access rights to see the requested data.

<!ELEMENT lockdiscovery (activelock)* >

13.8.1 Example - Retrieving the lockdiscovery Property

>>Request

```
PROPFIND /container/ HTTP/1.1
Host: www.foo.bar
Content-Length: xxxx
Content-Type: text/xml; charset=utf-8
<?xml version="1.0" encoding="utf-8" ?>
<D:propfind xmlns:D="DAV:">
  <D:prop><D:lockdiscovery/></D:prop>
</D:propfind>
```

>>Response

```
HTTP/1.1 207 Multi-Status
Content-Type: text/xml; Charset=utf-8
Content-Length: xxxx
<?xml version="1.0" encoding="utf-8" ?>
<D:multistatus xmlns:D="DAV:">
  <D:response>
    <D:href>http://www.foo.bar/container/</D:href>
    <D:propstat>
      <D:prop>
        <D:lockdiscovery>
          <D:activelock>
            <D:locktype><D:write/></D:locktype>
            <D:lockscope><D:exclusive/></D:lockscope>
            <D:depth><D:depth>
              <D:owner>Jane Smith</D:owner>
              <D:timeout>infinite</D:timeout>
            </D:locktoken>
            <D:href>
              <D:href>opaquelocktoken:f81de2ad-7f3d-alb2-4f3c-00a0c91a9d76
            </D:href>
          </D:activelock>
        </D:lockdiscovery>
      </D:prop>
      <D:status>HTTP/1.1 200 OK</D:status>
    </D:propstat>
  </D:response>
</D:multistatus>
```

This resource has a single exclusive write lock on it, with an infinite timeout.

13.9 resourcetype Property

Name: resourcetype
 Namespace: DAV:
 Purpose: Specifies the nature of the resource.
 Description: The resourcetype property MUST be defined on all DAV compliant resources. The default value is empty.

<ELEMENT resourcetype ANY >

13.10 source Property

Name: source
 Namespace: DAV:
 Purpose: The destination of the source link identifies the resource that contains the unprocessed source of the link's source.
 Description: The source of the link (src) is typically the URI of the output resource on which the link is defined, and there is typically only one destination (dst) of the link, which is the URI where the unprocessed source of the resource may be accessed. When more than one link destination exists, this specification asserts no policy on ordering.

<ELEMENT source (link)* >

13.10.1 Example - A source Property

```
<?xml version="1.0" encoding="utf-8" ?>
<D:prop xmlns:D="DAV:" xmlns:F="http://www.foo corp.com/Project/">
  <D:source>
    <D:link>
      <F:profiles>Source</F:profiles>
    </D:link>
    <D:link>
      <F:profiles>Library</F:profiles>
    </D:link>
    <D:link>
      <F:profiles>Makefile</F:profiles>
    </D:link>
  </D:source>
</D:prop>
```

In this example the resource http://foo.bar/program has a source property that contains three links. Each link contains three elements, two of which, src and dst, are part of the DAV schema defined in this document, and one which is defined by the schema http://www.foo corp.com/project/ (Source, Library, and Makefile). A client which only implements the elements in the DAV spec will not understand the foo corp elements and will ignore them, thus seeing the expected source and destination links. An enhanced client may know about the foo corp elements and be able to present the user with additional information about the links. This example demonstrates the power of XML markup, allowing element values to be enhanced without breaking older clients.

13.11 supportedlock Property

Name: supportedlock
 Namespace: DAV:
 Purpose: To provide a listing of the lock capabilities supported by the resource.
 Description: The supportedlock property of a resource returns a listing of the combinations of scope and access types which may be specified in a lock request on the resource. Note that the actual contents are themselves controlled by access controls so a server is not required to provide information the client is not authorized to see.

<ELEMENT supportedlock (lockentry)* >

14 Instructions for Processing XML in DAV

All DAV compliant resources MUST ignore any unknown XML element and all its children encountered while processing a DAV method that uses XML as its command language.

This restriction also applies to the processing, by clients, of DAV property values where unknown XML elements SHOULD be ignored unless the property's schema declares otherwise.

This restriction does not apply to setting dead DAV properties on the server where the server MUST record unknown XML elements.

Additionally, this restriction does not apply to the use of XML where XML happens to be the content type of the entity body, for example, when used as the body of a PUT.

Since XML can be transported as text/xml or application/xml, a DAV server MUST accept DAV method requests with XML parameters transported as either text/xml or application/xml, and DAV client MUST accept XML responses using either text/xml or application/xml.

15 DAV Compliance Classes

A DAV compliant resource can choose from two classes of compliance. A client can discover the compliance classes of a resource by executing OPTIONS on the resource, and examining the "DAV" header which is returned.

Since this document describes extensions to the HTTP/1.1 protocol, minimally all DAV compliant resources, clients, and proxies MUST be compliant with [RFC2068].

Compliance classes are not necessarily sequential. A resource that is class 2 compliant must also be class 1 compliant; but if additional compliance classes are defined later, a resource that is class 1, 2, and 4 compliant might not be class 3 compliant. Also note that identifiers other than numbers may be used as compliance class identifiers.

15.1 Class 1

A class 1 compliant resource MUST meet all "MUST" requirements in all sections of this document.

Class 1 compliant resources MUST return, at minimum, the value "1" in the DAV header on all responses to the OPTIONS method.

15.2 Class 2

A class 2 compliant resource MUST meet all class 1 requirements and support the LOCK method, the supportedlock property, the lockdiscovery property, the Time-Out response header and the Lock-Token request header. A class "2" compliant resource SHOULD also support the Time-Out request header and the owner XML element.

Class 2 compliant resources MUST return, at minimum, the values "1" and "2" in the DAV header on all responses to the OPTIONS method.

13.11.1 Example - Retrieving the supportedlock Property

```
>>Request
PROPFIND /container/ HTTP/1.1
Host: www.foo.bar
Content-Length: xxxx
Content-Type: text/xml; charset=utf-8

<?xml version="1.0" encoding="utf-8" ?>
<D:propfind xmlns:D="DAV:">
  <D:prop><D:supportedlock/></D:prop>
</D:propfind>

>>Response
HTTP/1.1 207 Multi-Status
Content-Type: text/xml; charset=utf-8
Content-Length: xxxx

<?xml version="1.0" encoding="utf-8" ?>
<D:multistatus xmlns:D="DAV:">
  <D:response>
    <D:href>http://www.foo.bar/container/</D:href>
    <D:propstat>
      <D:prop>
        <D:supportedlock>
          <D:lockentry>
            <D:lockscope><D:exclusive/></D:lockscope>
            <D:locktype><D:write/></D:locktype>
          </D:lockentry>
          <D:lockentry>
            <D:lockscope><D:shared/></D:lockscope>
            <D:locktype><D:write/></D:locktype>
          </D:lockentry>
        </D:supportedlock>
      </D:prop>
      <D:status>HTTP/1.1 200 OK</D:status>
    </D:propstat>
  </D:response>
</D:multistatus>
```

16 Internationalization Considerations

In the realm of internationalization, this specification complies with the IETF Character Set Policy [RFC2277]. In this specification, human-readable fields can be found either in the value of a property, or in an error message returned in a response entity body. In both cases, the human-readable content is encoded using XML, which has explicit provisions for character set tagging and encoding, and requires that XML processors read XML elements encoded, at minimum, using the UTF-8 [UTF-8] encoding of the ISO 10646 multilingual plane. XML examples in this specification demonstrate use of the charset parameter of the Content-Type header, as defined in [RFC2376], as well as the XML "encoding" attribute, which together provide charset identification information for MIME and XML processors.

XML also provides a language tagging capability for specifying the language of the contents of a particular XML element. XML uses either IANA registered language tags (see [RFC1766]) or ISO 639 language tags [ISO-639] in the "xml:lang" attribute of an XML element to identify the language of its content and attributes.

WebDAV applications MUST support the character set tagging, character set encoding, and the language tagging functionality of the XML specification. Implementors of WebDAV applications are strongly encouraged to read "XML Media Types" [RFC2376] for instruction on which MIME media type to use for XML transport, and on use of the charset parameter of the Content-Type header.

Names used within this specification fall into three categories: names of protocol elements such as methods and headers, names of XML elements, and names of properties. Naming of protocol elements follows the precedent of HTTP, using English names encoded in USASCII for methods and headers. Since these protocol elements are not visible to users, and are in fact simply long token identifiers, they do not need to support encoding in multiple character sets. Similarly, though the names of XML elements used in this specification are English names encoded in UTF-8, these names are not visible to the user, and hence do not need to support multiple character set encodings.

The name of a property defined on a resource is a URI. Although some applications (e.g., a generic property viewer) will display property URIs directly to their users, it is expected that the typical application will use a fixed set of properties, and will provide a mapping from the property name URI to a human-readable field when displaying the property name to a user. It is only in the case where the set of properties is not known ahead of time that an application need display a property name URI to a user. We recommend that applications provide human-readable property names wherever feasible.

For error reporting, we follow the convention of HTTP/1.1 status codes, including with each status code a short, English description of the code (e.g., 423 (Locked)). While the possibility exists that a poorly crafted user agent would display this message to a user, internationalized applications will ignore this message, and display an appropriate message in the user's language and character set.

Since interoperation of clients and servers does not require locale information, this specification does not specify any mechanism for transmission of this information.

17 Security Considerations

This section is provided to detail issues concerning security implications of which WebDAV applications need to be aware.

All of the security considerations of HTTP/1.1 (discussed in [RFC2068]) and XML (discussed in [RFC2376]) also apply to WebDAV. In addition, the security risks inherent in remote authoring require stronger authentication technology, introduce several new privacy concerns, and may increase the hazards from poor server design. These issues are detailed below.

17.1 Authentication of Clients

Due to their emphasis on authoring, WebDAV servers need to use authentication technology to protect not just access to a network resource, but the integrity of the resource as well. Furthermore, the introduction of locking functionality requires support for authentication.

A password sent in the clear over an insecure channel is an inadequate means for protecting the accessibility and integrity of a resource as the password may be intercepted. Since Basic authentication for HTTP/1.1 performs essentially clear text transmission of a password, Basic authentication MUST NOT be used to authenticate a WebDAV client to a server unless the connection is secure. Furthermore, a WebDAV server MUST NOT send Basic authentication credentials in a WWW-Authenticate header unless the connection is secure. Examples of secure connections include a Transport Layer Security (TLS) connection employing a strong cipher suite with mutual authentication of client and server, or a connection over a network which is physically secure, for example, an isolated network in a building with restricted access.

WebDAV applications MUST support the Digest authentication scheme [RFC2069]. Since Digest authentication verifies that both parties to a communication know a shared secret, a password, without having to send that secret in the clear, Digest authentication avoids the security problems inherent in Basic authentication while providing a level of authentication which is useful in a wide range of scenarios.

17.2 Denial of Service

Denial of service attacks are of special concern to WebDAV servers. WebDAV plus HTTP enables denial of service attacks on every part of a system's resources.

The underlying storage can be attacked by PUTting extremely large files.

Asking for recursive operations on large collections can attack processing time.

Making multiple pipelined requests on multiple connections can attack network connections.

WebDAV servers need to be aware of the possibility of a denial of service attack at all levels.

17.3 Security through Obscurity

WebDAV provides, through the PROPFIND method, a mechanism for listing the member resources of a collection. This greatly diminishes the effectiveness of security or privacy techniques that rely only on the difficulty of discovering the names of network resources. Users of WebDAV servers are encouraged to use access control techniques to prevent unwanted access to resources, rather than depending on the relative obscurity of their resource names.

17.4 Privacy Issues Connected to Locks

When submitting a lock request a user agent may also submit an owner XML field giving contact information for the person taking out the lock (for those cases where a person, rather than a robot, is taking out the lock). This contact information is stored in a lockdiscovery property on the resource,

and can be used by other collaborators to begin negotiation over access to the resource. However, in many cases this contact information can be very private, and should not be widely disseminated. Servers SHOULD limit read access to the lockdiscovery property as appropriate. Furthermore, user agents SHOULD provide control over whether contact information is sent at all, and if contact information is sent, control over exactly what information is sent.

17.5 Privacy Issues Connected to Properties

Since property values are typically used to hold information such as the author of a document, there is the possibility that privacy concerns could arise stemming from widespread access to a resource's property data. To reduce the risk of inadvertent release of private information via properties, servers are encouraged to develop access control mechanisms that separate read access to the resource body and read access to the resource's properties. This allows a user to control the dissemination of their property data without overly restricting access to the resource's contents.

17.6 Reduction of Security due to Source Link

HTTP/1.1 warns against providing read access to script code because it may contain sensitive information. Yet WebDAV, via its source link facility, can potentially provide a URI for script resources so they may be authored. For HTTP/1.1, a server could reasonably prevent access to source resources due to the predominance of read-only access. WebDAV, with its emphasis on authoring, encourages read and write access to source resources, and provides the source link facility to identify the source. This reduces the security benefits of eliminating access to source resources. Users and administrators of WebDAV servers should be very cautious when allowing remote authoring of scripts, limiting read and write access to the source resources to authorized principals.

17.7 Implications of XML External Entities

XML supports a facility known as "external entities", defined in section 4.2.2 of [REC-XML], which instruct an XML processor to retrieve and perform an inline include of XML located at a particular URI. An external XML entity can be used to append or modify the document type declaration (DTD) associated with an XML document. An external XML entity can also be used to include XML within the content of an XML document. For non-validating XML, such as the XML used in this specification, including an external XML entity is not required by [REC-XML]. However, [REC-XML] does state that an XML processor may, at its discretion, include the external XML entity.

External XML entities have no inherent trustworthiness and are subject to all the attacks that are endemic to any HTTP GET request. Furthermore, it is possible for an external XML entity to modify the DTD, and hence affect the final form of an XML document, in the worst case significantly modifying its semantics, or exposing the XML processor to the security risks discussed in [RFC2376]. Therefore, implementers must be aware that external XML entities should be treated as untrustworthy.

There is also the scalability risk that would accompany a widely deployed application which made use of external XML entities. In this situation, it is possible that there would be significant numbers of requests for one external XML entity, potentially overloading any server which fields requests for the resource containing the external XML entity.

17.8 Risks Connected with Lock Tokens

This specification, in section 6.4, requires the use of Universal Unique Identifiers (UUIDs) for lock tokens, in order to guarantee their uniqueness across space and time. UUIDs, as defined in [ISO-11578], contain a "node" field which "consists of the IEEE address, usually the host address. For systems with multiple IEEE 802 nodes, any available node address can be used." Since a WebDAV server will issue many locks over its lifetime, the implication is that it will also be publicly exposing its IEEE 802 address.

There are several risks associated with exposure of IEEE 802 addresses. Using the IEEE 802 address:

- It is possible to track the movement of hardware from subnet to subnet.
- It may be possible to identify the manufacturer of the hardware running a WebDAV server.
- It may be possible to determine the number of each type of computer running WebDAV. Section 6.4.1 of this specification details an alternate mechanism for generating the "node" field of a UUID without using an IEEE 802 address, which alleviates the risks associated with exposure of IEEE 802 addresses by using an alternate source of uniqueness.

18 IANA Considerations

This document defines two namespaces, the namespace of property names, and the namespace of WebDAV-specific XML elements used within property values.

URIs are used for both names, for several reasons. Assignment of a URI does not require a request to a central naming authority, and hence allow WebDAV property names and XML elements to be quickly defined by any WebDAV user or application. URIs also provide a unique address space, ensuring that the distributed users of WebDAV will not have collisions among the property names and XML elements they create.

This specification defines a distinguished set of property names and XML elements that are understood by all WebDAV applications. The property names and XML elements in this specification are all derived from the base URI DAV: by adding a suffix to this URI, for example, DAV:creationdate for the "creationdate" property.

This specification also defines a URI scheme for the encoding of lock tokens, the opaque-locktoken URI scheme described in section 6.4.

To ensure correct interpretation based on this specification, IANA must reserve the URI namespaces starting with "DAV:" and with "opaque-locktoken:" for use by this specification, its revisions, and related WebDAV specifications.

19 Intellectual Property

The following notice is copied from RFC 2026 [RFC2026], section 10.4, and describes the position of the IETF concerning intellectual property claims made against this document.

The IETF takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on the IETF's procedures with respect to rights in standards-track and standards-related documentation can be found in BCP-11. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementors or users of this specification can be obtained from the IETF Secretariat.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this standard. Please address the information to the IETF Executive Director.

20 Acknowledgements

A specification such as this thrives on piercing critical review and withers from apathetic neglect. The authors gratefully acknowledge the contributions of the following people, whose insights were so valuable at every stage of our work.

Terry Allen, Harald Alvestrand, Jim Amsden, Becky Anderson, Alan Babich, Sanford Barr, Dylan Barrell, Bernard Chesler, Tim Berners-Lee, Dan Connolly, Jim Cunningham, Ron Daniel, Jr., Jim Davis, Keith Dawson, Mark Day, Brian Deen, Martin Dierst, David Durand, Lee Farrell, Chuck Fay, Wesley Feller, Roy Fielding, Mark Fisher, Alan Freier, George Florentine, Jim Gettys, Phill Hallam-Baker, Dennis Hamilton, Steve Henning, Mead Himmelstein, Alex Hopmann, Andre van der Hoek, Ben Laurie, Paul Leach, Ora Lassila, Karen MacArthur, Steven Martin, Larry Masinter, Michael Mealling, Keith Moore, Thomas Narten, Henrik Nielsen, Kenji Ono, Bob Parker, Glenn Peterson, Jon Radoff, Saven Reddy, Henry Sanders, Christopher Seiwald, Judith Stein, Mike Spreitzer, Einar Stefferud, Greg Stein, Ralph Swick, Kenji Takahashi, Richard N. Taylor, Robert Thau, John Turner, Sankar Virdhagiswaran, Fabio Vitali, Gregory Woodhouse, and Lauren Wood.

Two from this list deserve special mention. The contributions by Larry Masinter have been invaluable, both in helping the formation of the working group and in patiently coaching the authors along the way. In so many ways he has set high standards we have toiled to meet. The contributions of Judith Stein in clarifying the requirements, and in patiently reviewing draft after draft, both improved this specification and expanded our minds on document management.

We would also like to thank John Turner for developing the XML DTD.

21 References

21.1 Normative References

[RFC1766] H. T. Alvestrand, "Tags for the Identification of Languages," RFC 1766. Uninett. March, 1995.

[RFC2277] H. T. Alvestrand, "IETF Policy on Character Sets and Languages," RFC 2277, BCP 18. Uninett. January, 1998.

[RFC2119] S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels," RFC 2119, BCP 14. Harvard University. March, 1997.

[RFC2396] T. Berners-Lee, R. Fielding, L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax," RFC 2396. MIT/LCS, U.C. Irvine, Xerox. August, 1998.

[REC-XML-JT] J. Bray, J. Paoli, C. M. Sperberg-McQueen, "Extensible Markup Language (XML)," World Wide Web Consortium Recommendation REC-xml-19980210. <http://www.w3.org/TR/1998/REC-xml-19980210>.

[REC-XML-NAMES] T. Bray, D. Hollander, A. Layman, "Name Spaces in XML," World Wide Web Consortium Recommendation REC-xml-names. <http://www.w3.org/TR/REC-xml-names-19990114/>

[RFC2069] J. Franks, P. Hallam-Baker, J. Hostetler, P. Leach, A. Luotonen, E. Sink, and L. Stewart, "An Extension to HTTP: Digest Access Authentication," RFC 2069. Northwestern University, CERN, Spyglass Inc., Microsoft Corp., Netscape Communications Corp., Spyglass Inc., Open Market Inc. January 1997.

[RFC2068] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee, "Hypertext Transfer Protocol - HTTP/1.1," RFC 2068. U.C. Irvine, DEC, MIT/LCS. January, 1997.

[ISO-639] ISO (International Organization for Standardization). ISO 639:1988. "Code for the representation of names of languages."

[ISO-8601] ISO (International Organization for Standardization), ISO 8601:1988. "Data elements and interchange formats - Information interchange - Representation of dates and times."

[ISO-11578] ISO (International Organization for Standardization). ISO/IEC 11578:1996. "Information technology - Open Systems Interconnection - Remote Procedure Call (RPC)"

[RFC2141] R. Moats, "URN Syntax," RFC 2141. AT&T. May, 1997.

[UTF-8] F. Yergeau, "UTF-8, a transformation format of Unicode and ISO 10646," RFC 2279. Alis Technologies. January, 1998.

22 Authors' Addresses

- Y. Y. Goland
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399
Email: yarong@microsoft.com
- E. J. Whitehead, Jr.
Dept. Of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
Email: ejw@ics.uci.edu
- A. Faizi
Netscape
685 East Middlefield Road
Mountain View, CA 94043
Email: asad@netscape.com
- S. R. Carter
Novell
1555 N. Technology Way
M/S ORM F111
Orem, UT 84097-2399
Email: scarter@novell.com
- D. Jensen
Novell
1555 N. Technology Way
M/S ORM F111
Orem, UT 84097-2399
Email: dcjensen@novell.com

21.2 Informational References

- [RFC2026] S. Bradner, "The Internet Standards Process - Revision 3," RFC 2026, BCP 9, Harvard University, October, 1996.
- [RFC1807] R. Lasher, D. Cohen, "A Format for Bibliographic Records," RFC 1807, Stamford, Myrtcom, June, 1995.
- [WF] C. Lagoze, "The Warwick Framework: A Container Architecture for Diverse Sets of Metadata," D-Lib Magazine, July/August 1996.
<http://www.dlib.org/dlib/july96/lagoze/07lagoze.html>
- [USMARC] Network Development and MARC Standards, Office, ed. 1994. "USMARC Format for Bibliographic Data", 1994, Washington, DC: Cataloging Distribution Service, Library of Congress.
- [REC-PICS] J. Miller, T. Krauskopf, P. Resnick, W. Treese, "PICS Label Distribution Label Syntax and Communication Protocols" Version 1.1, World Wide Web Consortium Recommendation REC-PICS-labels-961031. <http://www.w3.org/pub/WWW/TR/REC-PICS-labels-961031.html>.
- [RFC2291] J. A. Slein, F. Vitali, E. J. Whitehead, Jr., D. Durand, "Requirements for Distributed Authoring and Versioning Protocol for the World Wide Web," RFC 2291, Xerox, Univ. of Bologna, U.C. Irvine, Boston Univ, February, 1998.
- [RFC2413] S. Weibel, J. Kunze, C. Lagoze, M. Wolf, "Dublin Core Metadata for Resource Discovery," RFC 2413, OCLC, UCSF, Cornell, Reuters, September, 1998.
- [RFC2376] E. Whitehead, M. Murata, "XML Media Types," RFC 2376, U.C. Irvine, Fuji Xerox Info. Systems, July 1998.

23 Appendices

23.1 Appendix 1 - WebDAV Document Type Definition

This section provides a document type definition, following the rules in [REC-XML], for the XML elements used in the protocol stream and in the values of properties. It collects the element definitions given in sections 12 and 13.

```
<!DOCTYPE webdav-1.0 [
<!--===== XML Elements from Section 12 =====>
<ELEMENT activelock (lockscope, locktype, depth, owner?, timeout?,
locktoken?) >
<ELEMENT lockentry (lockscope, locktype) >
<ELEMENT lockinfo (lockscope, locktype, owner?) >
<ELEMENT locktype (write) >
<ELEMENT write EMPTY >
<ELEMENT lockscope (exclusive | shared) >
<ELEMENT exclusive EMPTY >
<ELEMENT shared EMPTY >
<ELEMENT depth (#PCDATA) >
<ELEMENT owner ANY >
<ELEMENT timeout (#PCDATA) >
<ELEMENT locktoken (href) >
<ELEMENT href (#PCDATA) >
<ELEMENT link (src*, dst*) >
<ELEMENT dst (#PCDATA) >
<ELEMENT src (#PCDATA) >
<ELEMENT multistatus (response*, responsesdescription?) >
<ELEMENT response (href, (href*, status) [(propstat*)),
responsesdescription?) >
<ELEMENT status (#PCDATA) >
<ELEMENT propstat (prop, status, responsesdescription?) >
<ELEMENT responsesdescription (#PCDATA) >
<ELEMENT prop ANY >
<ELEMENT propertybehavior (omit | keepalive) >
<ELEMENT omit EMPTY >
<ELEMENT keepalive (#PCDATA | href) >
<ELEMENT propertyupdate (remove | set)* >
<ELEMENT remove (prop) >
<ELEMENT set (prop) >
<ELEMENT propfind (allprop | proppatch | prop) >
<ELEMENT allprop EMPTY >
<ELEMENT proppatch EMPTY >
<ELEMENT collection EMPTY >
<!--===== Property Elements from Section 13 =====>
<ELEMENT creationdate (#PCDATA) >
<ELEMENT displayname (#PCDATA) >
```

```
<ELEMENT getcontentlength (#PCDATA) >
<ELEMENT getcontenttype (#PCDATA) >
<ELEMENT getetag (#PCDATA) >
<ELEMENT getlastmodified (#PCDATA) >
<ELEMENT lockdiscovery (activelock)* >
<ELEMENT resourcecset ANY >
<ELEMENT source (link)* >
<ELEMENT supportedlock (lockentry)* >
]>
```

23.2 Appendix 2 - ISO 8601 Date and Time Profile

The creationdate property specifies the use of the ISO 8601 date format [ISO-8601]. This section defines a profile of the ISO 8601 date format for use with this specification. This profile is quoted from an Internet-Draft by Chris Newman, and is mentioned here to properly attribute his work.

```
date-time      = full-date "T" full-time
full-date      = date-fullyear "-" date-month "-" date-day
full-time      = partial-time time-offset
date-fullyear  = 4DIGIT
date-month     = 2DIGIT ; 01-12
date-day       = 2DIGIT ; 01-28, 01-29, 01-30, 01-31 based on month/year
time-hour      = 2DIGIT ; 00-23
time-minute    = 2DIGIT ; 00-59
time-second    = 2DIGIT ; 00-59, 00-60 based on leap second rules
time-secfrac  = "." 1*DIGIT
time-numoffset = ("+" / "-") time-hour ":" time-minute
time-offset    = "Z" / time-numoffset
partial-time   = time-hour ":" time-minute ":" time-second
               [time-secfrac]
```

Numeric offsets are calculated as local time minus UTC (Coordinated Universal Time). So the equivalent time in UTC can be determined by subtracting the offset from the local time. For example, 18:50:00-04:00 is the same time as 22:58:00Z.

If the time in UTC is known, but the offset to local time is unknown, this can be represented with an offset of "-00:00". This differs from an offset of "Z" which implies that UTC is the preferred reference point for the specified time.

To understand why a 400 (Bad Request) is returned let us look at the request body as the server unfamiliar with expired-props sees it.

```
<?xml version="1.0" encoding="utf-8" ?>
<D:propfind xmlns:D="DAV:"
  xmlns:E="http://www.foo.bar/standards/props/">
  </D:propfind>
```

As the server does not understand the expired-props element, according to the WebDAV-specific XML-processing rules defined in section 14, it must ignore it. Thus the server sees an empty propfind, which by the definition of the propfind element is illegal.

Please note that had the extension been additive it would not necessarily have resulted in a 400 (Bad Request). For example, imagine the following request body for a PROPFIND:

```
<?xml version="1.0" encoding="utf-8" ?>
<D:propfind xmlns:D="DAV:"
  xmlns:E="http://www.foo.bar/standards/props/">
  <D:propname/>
  <E:leave-out>*boss*</E:leave-out>
</D:propfind>
```

The previous example contains the fictitious element leave-out. Its purpose is to prevent the return of any property whose name matches the submitted pattern. If the previous example were submitted to a server unfamiliar with leave-out, the only result would be that the leave-out element would be ignored and a propname would be executed.

23.4 Appendix 4 -- XML Namespaces for WebDAV

23.4.1 Introduction

All DAV compliant systems MUST support the XML namespace extension as specified in [REC-XML-NAMES].

23.4.2 Meaning of Qualified Names

[Note to the reader: This section does not appear in [REC-XML-NAMES], but is necessary to avoid ambiguity for WebDAV XML processors.]

WebDAV compliant XML processors MUST interpret a qualified name as a URI constructed by appending the LocalPart to the namespace name URI.

Example

```
<del:glider xmlns:del="http://www.del.jensen.org/">
  <del:glidername>
    Johnny Updraft
  </del:glidername>
  <del:glideraccidents/>
</del:glider>
```

In this example, the qualified element name "del:glider" is interpreted as the URL "http://www.del.jensen.org/del:glider".

```
<bar:glider xmlns:del="http://www.del.jensen.org/">
  <bar:glidername>
    Johnny Updraft
  </bar:glidername>
  <bar:glideraccidents/>
</bar:glider>
```

23.3 Appendix 3 - Notes on Processing XML Elements

23.3.1 Notes on Empty XML Elements

XML supports two mechanisms for indicating that an XML element does not have any content. The first is to declare an XML element of the form <A>. The second is to declare an XML element of the form <A/>. The two XML elements are semantically identical.

It is a violation of the XML specification to use the <A> form if the associated DTD declares the element to be EMPTY (e.g., <ELEMENT A EMPTY>). If such a statement is included, then the empty element format, <A/> must be used. If the element is not declared to be EMPTY, then either form <A> or <A/> may be used for empty elements.

23.3.2 Notes on Illegal XML Processing

XML is a flexible data format that makes it easy to submit data that appears legal but in fact is not. The philosophy of "Be flexible in what you accept and strict in what you send" still applies, but it must not be applied inappropriately. XML is extremely flexible in dealing with issues of white space, element ordering, inserting new elements, etc. This flexibility does not require extension, especially not in the area of the meaning of elements.

There is no kindness in accepting illegal combinations of XML elements. At best it will cause an unwanted result and at worst it can cause real damage.

23.3.2.1 Example - XML Syntax Error

The following request body for a PROPFIND method is illegal.

```
<?xml version="1.0" encoding="utf-8" ?>
<D:propfind xmlns:D="DAV:"
  <D:allprop/>
  <D:propname/>
</D:propfind>
```

The definition of the propfind element only allows for the allprop or the propname element, not both. Thus the above is an error and must be responded to with a 400 (Bad Request).

Imagine, however, that a server wanted to be "kind" and decided to pick the allprop element as the true element and respond to it. A client running over a bandwidth limited line who intended to execute a propname would be in for a big surprise if the server treated the command as an allprop.

Additionally, if a server were lenient and decided to reply to this request, the results would vary randomly from server to server, with some servers executing the allprop directive, and others executing the propname directive. This reduces interoperability rather than increasing it.

23.3.2.2 Example - Unknown XML Element

The previous example was illegal because it contained two elements that were explicitly banned from appearing together in the propfind element. However, XML is an extensible language, so one can imagine new elements being defined for use with propfind. Below is the request body of a PROPFIND and, like the previous example, must be rejected with a 400 (Bad Request) by a server that does not understand the expired-props element.

```
<?xml version="1.0" encoding="utf-8" ?>
<D:propfind xmlns:D="DAV:"
  xmlns:E="http://www.foo.bar/standards/props/">
  <E:expired-props/>
</D:propfind>
```

Even though this example is syntactically different from the previous example, it is semantically identical. Each instance of the namespace name "bar" is replaced with "http://www.del.jensen.org/" and then appended to the local name for each element tag. The resulting tag names in this example are exactly the same as for the previous example.

```
<foo:r xmlns:foo="http://www.del.jensen.org/glide">
  <foo:rname>
    Johnny Update
  </foo:rname>
  <foo:raccidents/>
</foo:r>
```

This example is semantically identical to the two previous ones. Each instance of the namespace name "foo" is replaced with "http://www.del.jensen.org/glide" which is then appended to the local name for each element tag, the resulting tag names are identical to those in the previous examples.

24 Full Copyright Statement

Copyright (C) The Internet Society (1999). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

IEC 61834-4
JTSB-508-US (5)

INTERNATIONAL STANDARD

IEC 61834-4

First edition
1998-07

Recording –
Helical-scan digital video cassette recording system
using 6,35 mm magnetic tape for consumer use
(525-60, 625-50, 1125-60 and 1250-50 systems) –

Part 4:
Pack header table and contents

Enregistrement –
Système d'enregistrement grand public vidéo à cassette
à défilement hélicoïdal pour bande magnétique de 6,35 mm
(systèmes 525-60, 625-50, 1125-60 et 1250-50)

Partie 4:
Tableaux des paquets en-tête et leur contenu

© IEC 1998 — Copyright - all rights reserved

No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

International Electrotechnical Commission
Telefax: +41 22 919 0300

3, rue de Varembeé Geneva, Switzerland
e-mail: inmail@iec.ch IEC web site <http://www.iec.ch>



Commission Electrotechnique Internationale
International Electrotechnical Commission
Международная Электротехническая Комиссия

PRICE CODE XD

For price, see current catalogue

CONTENTS

	Page
FOREWORD	6
Clause	
1 General.....	8
1.1 Scope	8
1.2 Normative references	8
1.3 Labelling convention.....	8
2 Pack header table	9
3 CONTROL	11
3.1 CASSETTE ID.....	11
3.2 TAPE LENGTH	12
3.3 TIMER ACT DATE (Timer activation date)	13
3.4 TIMER ACT S/S (Timer activation start/stop)	17
3.5 PR START POINT (Playback or recording start point).....	18
3.6 PR START POINT (Playback or recording start point).....	19
3.7 TAG ID NO. / GENRE	20
3.8 TOPIC/PAGE HEADER	27
3.9 TEXT HEADER	29
3.10 TEXT	32
3.11 TAG.....	33
3.12 TAG.....	34
3.13 TELETEXT INFO (Teletext information).....	36
3.14 KEY	38
3.15 ZONE END	39
3.16 ZONE END	40
4 TITLE.....	42
4.1 TOTAL TIME.....	42
4.2 REMAIN TIME.....	43
4.3 CHAPTER TOTAL NO.	44
4.4 TIME CODE	45
4.5 BINARY GROUP	47
4.6 CASSETTE NO.	47
4.7 SOFT ID	48
4.8 SOFT ID	49
4.9 TEXT HEADER	50
4.10 TEXT	51
4.11 TITLE START	52
4.12 TITLE START	53
4.13 REEL ID.....	54
4.14 REEL ID.....	54
4.15 TITLE END	55
4.16 TITLE END	56

Clause

5	CHAPTER.....	57
5.1	TOTAL TIME.....	57
5.2	REMAIN TIME.....	58
5.3	CHAPTER NO.	59
5.4	TIME CODE.....	60
5.5	BINARY GROUP.....	61
5.6	Reserved.....	61
5.7	Reserved.....	62
5.8	Reserved.....	62
5.9	TEXT HEADER.....	63
5.10	TEXT.....	64
5.11	CHAPTER START.....	65
5.12	CHAPTER START.....	66
5.13	Reserved.....	67
5.14	Reserved.....	67
5.15	CHAPTER END.....	68
5.16	CHAPTER END.....	69
6	PART.....	70
6.1	TOTAL TIME.....	70
6.2	REMAIN TIME.....	71
6.3	PART NO.	72
6.4	TIME CODE.....	73
6.5	BINARY GROUP.....	74
6.6	Reserved.....	74
6.7	Reserved.....	75
6.8	Reserved.....	75
6.9	TEXT HEADER.....	76
6.10	TEXT.....	77
6.11	PART START.....	78
6.12	PART START.....	79
6.13	Reserved.....	80
6.14	Reserved.....	80
6.15	PART END.....	81
6.16	PART END.....	82
7	PROGRAMME.....	83
7.1	TOTAL TIME.....	83
7.2	REMAIN TIME.....	84
7.3	REC DTIME (REC DATE/TIME).....	85
7.4	TIME CODE.....	87
7.5	BINARY GROUP.....	88
7.6	Reserved.....	88
7.7	Reserved.....	89
7.8	Reserved.....	89
7.9	TEXT HEADER.....	90

Clause	Page
7.10 TEXT	91
7.11 PROGRAMME START	92
7.12 PROGRAMME START	93
7.13 Reserved	94
7.14 Reserved	94
7.15 PROGRAMME END	95
7.16 PROGRAMME END	96
8 AAUX.....	97
8.1 SOURCE	97
8.2 SOURCE CONTROL	101
8.3 REC DATE	104
8.4 REC TIME	105
8.5 BINARY GROUP	106
8.6 CLOSED CAPTION	107
8.7 TR (Transparent)	108
8.8 Reserved	108
8.9 TEXT HEADER	109
8.10 TEXT	110
8.11 AAUX START	111
8.12 AAUX START	112
8.13 Reserved	113
8.14 Reserved	113
8.15 AAUX END	114
8.16 AAUX END	115
9 VAUX.....	116
9.1 SOURCE	116
9.2 SOURCE CONTROL	120
9.3 REC DATE (Recording date)	130
9.4 REC TIME	132
9.5 BINARY GROUP	133
9.6 CLOSED CAPTION	133
9.7 TR (Transparent)	134
9.8 TELETEXT	135
9.9 TEXT HEADER	137
9.10 TEXT	138
9.11 VAUX START	139
9.12 VAUX START	140
9.13 MARINE/MOUNTAIN	141
9.14 LONGITUDE/LATITUDE	144
9.15 VAUX END	146
9.16 VAUX END	147

Clause	Page
10 CAMERA.....	148
10.1 CONSUMER CAMERA 1	148
10.2 CONSUMER CAMERA 2	150
10.3 Reserved	152
10.4 LENS	153
10.5 GAIN	154
10.6 PEDESTAL	155
10.7 GAMMA	156
10.8 DETAIL	157
10.9 TEXT HEADER	158
10.10 TEXT	159
10.11 Reserved	159
10.12 CAMERA PRESET	160
10.13 FLARE	162
10.14 SHADING	163
10.15 KNEE	165
10.16 SHUTTER	166
11 LINE	167
11.1 LINE HEADER	167
11.2 Y	171
11.3 CR	171
11.4 CB	172
11.5 Reserved	172
11.6 Reserved	173
11.7 Reserved	173
11.8 Reserved	174
11.9 TEXT HEADER	175
11.10 TEXT	176
11.11 LINE START	177
11.12 LINE START	178
11.13 Reserved	179
11.14 Reserved	179
11.15 LINE END	180
11.16 LINE END	181
12 SOFT MODE	182
12.1 MAKER CODE	182
12.2 OPTION	183
12.3 OPTION	183
12.4 OPTION	184
12.5 OPTION	184
12.6 OPTION	185
12.7 OPTION	185
12.8 OPTION	186
12.9 OPTION	186
12.10 OPTION	187
12.11 OPTION	187
12.12 OPTION	188
12.13 OPTION	188
12.14 OPTION	189
12.15 OPTION	189
12.16 NO INFO: No information	189

INTERNATIONAL ELECTROTECHNICAL COMMISSION

RECORDING – HELICAL-SCAN DIGITAL VIDEO CASSETTE RECORDING SYSTEM USING 6,35 mm MAGNETIC TAPE FOR CONSUMER USE (525-60, 625-50, 1125-60 and 1250-50 systems) –

Part 4: Pack header table and contents

FOREWORD

- 1) The IEC (International Electrotechnical Commission) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of the IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, the IEC publishes International Standards. Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. The IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
- 2) The formal decisions or agreements of the IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested National Committees.
- 3) The documents produced have the form of recommendations for international use and are published in the form of standards, technical reports or guides and they are accepted by the National Committees in that sense.
- 4) In order to promote international unification, IEC National Committees undertake to apply IEC International Standards transparently to the maximum extent possible in their national and regional standards. Any divergence between the IEC Standard and the corresponding national or regional standard shall be clearly indicated in the latter.
- 5) The IEC provides no marking procedure to indicate its approval and cannot be rendered responsible for any equipment declared to be in conformity with one of its standards.
- 6) Attention is drawn to the possibility that some of the elements of this International Standard may be the subject of patent rights. The IEC shall not be held responsible for identifying any or all such patent rights.

International Standard IEC 61834-4 has been prepared by subcommittee 100B: Audio, video and multimedia information storage systems, of IEC technical committee 100: Audio, video and multimedia systems and equipment.

The text of this standard is based on the following documents:

FDIS	Report on voting
100B/164/FDIS	100B/174/RVD

Full information on the voting for the approval of this standard can be found in the report on voting indicated in the above table.

IEC 61834 consists of the following parts:

- Part 1: General specifications;
- Part 2: SD format for 525-60 and 625-50 systems;
- Part 3: HD format for 1125-60 and 1250-50 systems;
- Part 4: Pack header table and contents;
- Part 5: Character information system.

This document is Part 4 of IEC 61834 and describes the pack header table and the contents of packs which are applicable to the whole recording system of helical-scan digital video cassette.

Part 1 describes the common specifications for the helical-scan digital video cassette recording system using 6,35 mm magnetic tape.

Part 2 describes the specifications for 525-60 and 625-50 systems which are not included in Part 1.

Part 3 describes the specifications for 1125-60 and 1250-50 systems which are not included in Part 1 and Part 2.

Part 5 describes the character information system which is applicable to the whole recording system of helical-scan digital video cassette.

For manufacturing SD digital video cassette recording system, Part 1, Part 2, Part 4 and Part 5 are referred to.

For manufacturing HD digital video cassette recording system, Part 1, Part 2, Part 3, Part 4 and Part 5 are referred to.

This part of IEC 61834 is to be referred to particularly when the pack header table and the contents are to be checked.

A bilingual version of this standard may be issued at a later date.

9 VAUX

VAUX 0

9.1 SOURCE

MSB					LSB					
PC 0	0	1	1	0	0	0	0	0		
PC 1	TENS of TV CHANNEL				UNITS of TV CHANNEL					
PC 2	B/W	EN	CLF		HUNDREDS of TV CHANNEL					
PC 3	SOURCE CODE		50/60	STYPE						
PC 4	TUNER CATEGORY									

This pack shall be recorded at least in the VAUX main area.

TV CHANNEL: The number of the television channel

001 to 999 = Television channel

EEEh = Pre-recorded tape or LINE (MUSE)

FFFh = No information

TV CHANNEL should indicate the channel number which is assigned to the broadcasting station, and it may indicate the channel number which is set by the user on the receiver.

B/W: Black and white flag

0 = Black and white

1 = Colour

B/W flag should be set to 1 for consumer digital VCR.

EN: Colour frames enable flag

0 = CLF is valid

1 = CLF is invalid

CLF: Colour frames identification code (refer to ITU-R Report 624-4)

For 525-60 system

00b = Colour frame A

01b = Colour frame B

Others = reserved

For 625-50 system

00b = 1st, 2nd field

01b = 3rd, 4th field

10b = 5th, 6th field

11b = 7th, 8th field

50/60:

0 = 60 field system

1 = 50 field system

The specification of B/W, EN, 50/60 and CLF

B/W	EN	50/60	CLF	System	Colour frame	
1	0	0	00	525-60	Valid	Colour frame A
			01			Colour frame B
		1	00	625-50		1st, 2nd fields
			01			3rd, 4th fields
			10			5th, 6th fields
11	7th, 8th fields					
X	1	X	11		Invalid	

X don't care

SOURCE CODE:

SOURCE CODE defines the input source of the video signal in combination with TV CHANNEL and TUNER CATEGORY as follows.

SOURCE CODE	TV CHANNEL			TUNER CATEGORY	Input source
	100's	10's	1's		
00	Fh	Fh	Fh	FFh	Camera
01	Eh	Eh	Eh	FFh	Line (MUSE)
01	Fh	Fh	Fh	FFh	Line
10	0h	0h	1h	FFh	Cable Ch1
	0h	0h	2h		Ch2
	9h	9h	9h		Ch999
11	0h	0h	1h	Prescribed value	Tuner Ch1
	0h	0h	2h		Ch2
	9h	9h	9h		Ch999
11	Eh	Eh	Eh	FFh	Pre-recorded tape
11	Fh	Fh	Fh	FFh	No information

STYPE:

STYPE defines a signal type of video signal in combination with the 50/60 flag as follows.

STYPE	50/60	
	0	1
00000	525-60 system	625-50 system
00001	Reserved	
00010	1125-60 system	1250-50 system
00011	Reserved	
...		
...		
11111		

TUNER CATEGORY:

TUNER CATEGORY consists of area number and satellite number as follows.

TUNER CATEGORY = FFh is indicative of no information.

Area number				Satellite number			
b7	b6	b5	b4	b3	b2	b1	b0

Area number specification

Area number	Region	Area
0 0 0	Region 1	Europe, Africa
0 0 1		
0 1 0	Region 2	North America, South America
0 1 1		
1 0 0		
1 0 1		
1 1 0	Region 3	Asia, Oceania
1 1 1		

Details of area number are to be decided.

For region 1

Area number	Satellite number	Satellite name
0 0 0	0 0 0 0 0	UHF/VHF
	0 0 0 0 1	Reserved
	0 0 0 1 0	ASTRA A+B
	0 0 0 1 1	ASTRA C+D
	0 0 1 0 0	TELECOM (France)
	0 0 1 0 1	TELECOM-2
	0 0 1 1 0	Reserved
	1 1 1 1 1	
0 0 1	0 0 0 0 0	UHF/VHF
	0 0 0 0 1	Reserved
	1 1 1 1 1	

For region 2

Area number	Satellite number	Satellite name
0 1 0	0 0 0 0 0	UHF/VHF
	0 0 0 0 1	Reserved
	1 1 1 1 1	
0 1 1	0 0 0 0 0	UHF/VHF
	0 0 0 0 1	Reserved
	1 1 1 1 1	
1 0 0	0 0 0 0 0	UHF/VHF
	0 0 0 0 1	Reserved
	1 1 1 1 1	
1 0 1	0 0 0 0 0	UHF/VHF
	0 0 0 0 1	Reserved
	1 1 1 1 1	

For region 3

Area number	Satellite number	Satellite name
1 1 0	0 0 0 0 0	UHF/VHF
	0 0 0 0 1	BS
	0 0 0 1 0	SCC-A
	0 0 0 1 1	SCC-B
	0 0 1 0 0	JCSAT-1
	0 0 1 0 1	JCSAT-2
	0 0 1 1 0	Reserved
	1 1 1 1 1	
1 1 1	0 0 0 0 0	UHF/VHF
	0 0 0 0 1	Reserved
	1 1 1 1 0	

VAUX 1

9.2 SOURCE CONTROL

MSB										LSB									
PC 0	0		1		1		0		0		0		0		1				
PC 1	CGMS				ISR				CMP				SS						
PC 2	REC ST		1		REC MODE				1		DISP								
PC 3	FF		FS		FC		IL		ST		SC		BCSYS						
PC 4	1		GENRE CATEGORY																

This pack shall be recorded at least in the VAUX main area.

CGMS: Copy generation management system

00b = Copying permitted without restriction

01b = Not used

10b = One generation of copying permitted

11b = No copying permitted

If CGMS information encoded in the incoming signal is "0 0", a digital VCR may make a copy and shall encode "0 0", on "CGMS".

If CGMS information encoded in the incoming signal is "1 0", a digital VCR may make a copy and shall encode "1 1", on "CGMS".

If CGMS information encoded in the incoming signal is "1 1", a digital VCR shall not make a copy.

Each manufacturer has the discretion to follow the rules described above unless there is any legislation or similar mandating this.

ISR: Input source of just previous recording

00b = Analogue input

01b = Digital input

10b = Reserved

11b = No information

CMP: The number of times of compression

00b = Compression once

01b = Compression twice

10b = Compression three times or more

11b = No information

SS: Source and recorded situation

00b = Scrambled source with audience restrictions
and recorded without descrambling

01b = Scrambled source without audience restrictions
and recorded without descrambling

10b = Source with audience restrictions
or descrambled source with audience restrictions

11b = No information

If SS = 10b, then KEY pack should be recorded in the VAUX common optional area.

REC ST: Recording start point

0 = Recording start point

1 = Not recording start point

The duration of recording start point should be the period of 30 frames (525-60 system) or 25 frames (625-50 system).

REC MODE:

00b = Original

01b = Reserved

10b = Insert

11b = Invalid recording

where

Original: Video and two audio blocks are recorded simultaneously.

Insert: Video area is recorded with the pre-recorded audio blocks remaining as they are.

Invalid recording: Recorded video data are not taken into account.

BCSYS: Broadcast system

BCSYS indicates the type information of display format with DISP.

00b = type 0 (refer to IEC 61880, EIA-608)

01b = type 1 (refer to prETS 300 294)

Others = Reserved

DISP: Display select mode

BCSYS	DISP	Aspect ratio and format	Position
00	000	4 : 3 full format	Not applicable
	001	16 : 9 letter box	Centre
	010	16 : 9 full format (squeeze)	Not applicable
	011	Reserved	
	...		
	111		
01	000	4 : 3 full format	Not applicable
	001	14 : 9 letter box	Centre
	010	14 : 9 letter box	Top
	011	16 : 9 letter box	Centre
	100	16 : 9 letter box	Top
	101	> 16 : 9 letter box	Centre
	110	14 : 9 full format	Centre
	111	16 : 9 full format (anamorphic)	Not applicable
10	000	Reserved	
...	...		
11	111		

FF: Frame/Field flag

FF indicates whether both fields are output in order or only one of them is output twice during one frame period.

0 = Only one of two fields is output twice

1 = Both fields are output in order

FS: First/Second flag

FS indicates a field which should be output during field 1 period.

0 = Field 2 is output

1 = Field 1 is output

FF	FS	Output field
1	1	Field 1 and field 2 are output in this order
1	0	Field 2 and field 1 are output in this order
0	1	Field 1 is output twice
0	0	Field 2 is output twice

FC: Frame change flag

FC indicates whether the picture of the current frame is the same picture of the immediate previous frame.

0 = Same picture as the immediate previous frame

1 = Different picture from the immediate previous frame

IL: Interlace flag

IL indicates whether the data of two fields which construct one frame are interlaced or non-interlaced.

0 = Non-interlaced

1 = Interlaced or unrecognized

ST: Still-field picture flag

ST indicates the time difference between the two fields within a frame. This flag shall have the same value for a duration of at least three frames.

0 = The time difference between the fields is approximately 0 s.

1 = The time difference between the fields is approximately 1,001/60 s (525-60 system) or approximately 1/50 s (625-50 system).

SC: Still camera picture flag

This flag is prepared for distinguishing a still camera picture. Still camera picture:

Consecutive five frame of the same picture. For SC = 0, this flag may be used for displaying a still camera picture by stopping tape travelling automatically.

0 = Still camera picture

1 = Not still camera picture

GENRE CATEGORY:

GENRE CATEGORY shows the category of the video source.

The details are described in TIMER ACT DATE pack.

There are four types of input video signals:

- interlaced motion picture: a normal standard TV signal;
- non-interlaced motion picture: a non-interlaced TV signal in a frame like a video game output;
- frame still picture: a still picture during a frame and the still picture is an interlace TV signal in a frame;
- field still picture: a still picture during a field and the same still picture is repeated twice in a frame.

If the type of an input signal is indefinite, interlaced motion picture should be selected.

Recording frames		a		b		c		d		e	
Recording fields		a1	a2	b1	b2	c1	c2	d1	d2	e1	e2
Interlaced motion picture	FF	1		1		1		1		1	
	FS	1		1		1		1		1	
	FC	1		1		1		1		1	
	IL	1		1		1		1		1	
	ST	1		1		1		1		1	
Non-interlaced motion picture	FF	1		1		1		1		1	
	FS	1		1		1		1		1	
	FC	1		1		1		1		1	
	IL	0		0		0		0		0	
	ST	1		1		1		1		1	
Frame still picture	FF	1		1		1		1		1	
	FS	1		1		1		1		1	
	FC	0		1		0		0		1	
	IL	1		1		1		1		1	
	ST	0		0		0		0		0	
Field still picture	FF	1		1		1		1		1	
	FS	1		1		1		1		1	
	FC	0		1		0		0		1	
	IL	0		0		0		0		0	
	ST	0		0		0		0		0	

NOTE – For frame still pictures and field still pictures, frames b, c and d are still frames and have the same frame data.

For normal playback

Reproducing frames from tape		a		b		c		d		e	
Reproducing fields from tape		a1	a2	b1	b2	c1	c2	d1	d2	e1	e2
Interlaced motion picture	OT ¹⁾	a1	a2	b1	b2	c1	c2	d1	d2	e1	e2
	OD ²⁾	a1	a2	b1	b2	c1	c2	d1	d2	e1	e2
	FF	1		1		1		1		1	
	FS	1		1		1		1		1	
	FC	1		1		1		1		1	
	IL	1		1		1		1		1	
	ST	1		1		1		1		1	
Non-interlaced motion picture	OT	a1	a2	b1	b2	c1	c2	d1	d2	e1	e2
	OD	a1	a2	b1	b2	c1	c2	d1	d2	e1	e2
	FF	1		1		1		1		1	
	FS	1		1		1		1		1	
	FC	1		1		1		1		1	
	IL	0		0		0		0		0	
	ST	1		1		1		1		1	
Frame still picture	OT	a1	a2	b1	b2	c1	c2	d1	d2	e1	e2
	OD	a1	a2	b1	b2	c1	c2	d1	d2	e1	e2
	FF	1		1		1		1		1	
	FS	1		1		1		1		1	
	FC	0		1		0		0		1	
	IL	1		1		1		1		1	
	ST	0		0		0		0		0	
Field still picture	OT	a1	a2	b1	b2	c1	c2	d1	d2	e1	e2
	OD	a1	a2	b1	b2	c1	c2	d1	d2	e1	e2
	FF	1		1		1		1		1	
	FS	1		1		1		1		1	
	FC	0		1		0		0		1	
	IL	0		0		0		0		0	
	ST	0		0		0		0		0	

1) OT output order to the TV screen
2) OD output order to the digital interface

NOTE – For frame still pictures and field still pictures, frames b, c and d are still frames and have the same frame data.

Reproducing frames from tape		a		b		b		b		c	
Reproducing fields from tape		a1	a2	b1	b2	b1	b2	b1	b2	c1	c2
Interlaced motion picture (field slow)	OT ¹⁾	a2	a2	b1	b1	b1	b2	b2	b2	c1	c1
	OD ²⁾	a1	a2	b1	b2	b1	b2	b1	b2	c1	c2
	FF	0		0		1		0		0	
	FS	0		1		1		0		1	
	FC	0		1		0		0		1	
	IL	1		1		1		1		1	
	ST	1		1		1		1		1	
Interlaced motion picture (frame slow)	OT	a1	a2	b1	b2	b1	b2	b1	b2	c1	c2
	OD	a1	a2	b1	b2	b1	b2	b1	b2	c1	c2
	FF	1		1		1		1		1	
	FS	1		1		1		1		1	
	FC	0		1		0		0		1	
	IL	1		1		1		1		1	
	ST	1		1		1		1		1	
Non-interlaced motion picture (field slow)	OT	a2	a2	b1	b1	b1	b2	b2	b2	c1	c1
	OD	a1	a2	b1	b2	b1	b2	b1	b2	c1	c2
	FF	0		0		1		0		0	
	FS	0		1		1		0		1	
	FC	0		1		0		0		1	
	IL	0		0		0		0		0	
	ST	1		1		1		1		1	
Frame still picture	OT	a1	a2	b1	b2	b1	b2	b1	b2	c1	c2
	OD	a1	a2	b1	b2	b1	b2	b1	b2	c1	c2
	FF	1		1		1		1		1	
	FS	1		1		1		1		1	
	FC	0		1		0		0		0	
	IL	1		1		1		1		1	
	ST	0		0		0		0		0	
Field still picture	OT	a1	a2	b1	b2	b1	b2	b1	b2	c1	c2
	OD	a1	a2	b1	b2	b1	b2	b1	b2	c1	c2
	FF	1		1		1		1		1	
	FS	1		1		1		1		1	
	FC	0		1		0		0		0	
	IL	0		0		0		0		0	
	ST	0		0		0		0		0	

1) OT output order to the TV screen.
2) OD output order to the digital interface.

NOTE - For frame still pictures and field still pictures, frames b, c and d are still frames and have the same frame data.

Reproducing frames from tape		e		d		d		d		c	
Reproducing fields from tape		e1	e2	d1	d2	d1	d2	d1	d2	c1	c2
Interlaced motion picture (field slow)	OT ¹⁾	e1	e1	d2	d2	d2	d1	d1	d1	c2	c2
	OD ²⁾	e1	e2	d1	d2	d1	d2	d1	d2	c1	c2
	FF	0		0		1		0		0	
	FS	1		0		0		1		0	
	FC	0		1		0		0		1	
	IL	1		1		1		1		1	
	ST	1		1		1		1		1	
Interlaced motion picture (frame slow)	OT	e1	e2	d1	d2	d1	d2	d1	d2	c1	c2
	OD	e1	e2	d1	d2	d1	d2	d1	d2	c1	c2
	FF	1		1		1		1		1	
	FS	1		1		1		1		1	
	FC	0		1		0		0		1	
	IL	1		1		1		1		1	
	ST	1		1		1		1		1	
Non-interlaced motion picture (field slow)	OT	e1	e1	d2	d2	d2	d1	d1	d1	c2	c2
	OD	e1	e2	d1	d2	d1	d2	d1	d2	c1	c2
	FF	0		0		1		0		0	
	FS	1		0		0		1		0	
	FC	0		1		0		0		1	
	IL	0		0		0		0		0	
	ST	1		1		1		1		1	
Frame still picture	OT	e1	e2	d1	d2	d1	d2	d1	d2	c1	c2
	OD	e1	e2	d1	d2	d1	d2	d1	d2	c1	c2
	FF	1		1		1		1		1	
	FS	1		1		1		1		1	
	FC	0		1		0		0		0	
	IL	1		1		1		1		1	
	ST	0		0		0		0		0	
Field still picture	OT	e1	e2	d1	d2	d1	d2	d1	d2	c1	c2
	OD	e1	e2	d1	d2	d1	d2	d1	d2	c1	c2
	FF	1		1		1		1		1	
	FS	1		1		1		1		1	
	FC	0		1		0		0		0	
	IL	0		0		0		0		0	
	ST	0		0		0		0		0	

1) OT output order to the TV screen

2) OD output order to the digital interface

NOTE - For frame still pictures and field still pictures, frames b, c and d are still frames and have the same frame data.

For still playback

Reproducing frames from tape		a		b		b		b		b	
Reproducing fields from tape		a1	a2	b1	b2	b1	b2	b1	b2	b1	b2
Interlaced motion picture (field still type 1)	OT ¹⁾	a1	a2	b1	b1	b1	b1	b1	b1	b1	b1
	OD ²⁾	a1	a2	b1	b2	b1	b2	b1	b2	b1	b2
	FF	1		0		0		0		0	
	FS	1		1		1		1		1	
	FC	1		1		0		0		0	
	IL	1		1		1		1		1	
	ST	1		1		1		1		1	
Interlaced motion picture (field still type 2)	OT	a1	a2	b1	b2	b2	b2	b2	b2	b2	b2
	OD	a1	a2	b1	b2	b1	b2	b1	b2	b1	b2
	FF	1		1		0		0		0	
	FS	1		1		0		0		0	
	FC	1		1		0		0		0	
	IL	1		1		1		1		1	
	ST	1		1		1		1		1	
Interlaced motion picture (frame still)	OT	a1	a2	b1	b2	b1	b2	b1	b2	b1	b2
	OD	a1	a2	b1	b2	b1	b2	b1	b2	b1	b2
	FF	1		1		1		1		1	
	FS	1		1		1		1		1	
	FC	1		1		0		0		0	
	IL	1		1		1		1		1	
	ST	1		1		1		1		1	
Non-interlaced motion picture (field still type 1)	OT	a1	a2	b1	b1	b1	b1	b1	b1	b1	b1
	OD	a1	a2	b1	b2	b1	b2	b1	b2	b1	b2
	FF	1		0		0		0		0	
	FS	1		1		1		1		1	
	FC	1		1		0		0		0	
	IL	0		0		0		0		0	
	ST	1		1		1		1		1	
Non-interlaced motion picture (field still type 2)	OT	a1	a2	b1	b2	b2	b2	b2	b2	b2	b2
	OD	a1	a2	b1	b2	b1	b2	b1	b2	b1	b2
	FF	1		1		0		0		0	
	FS	1		1		0		0		0	
	FC	1		1		0		0		0	
	IL	0		0		0		0		0	
	ST	1		1		1		1		1	

1) OT output order to the TV screen
2) OD output order to the digital interface

For still playback (concluded)

Reproducing frames from tape		a		b		b		b		b	
Reproducing fields from tape		a1	a2	b1	b2	b1	b2	b1	b2	b1	b2
Frame still picture	OT ¹⁾	a1	a2	b1	b2	b1	b2	b1	b2	b1	b2
	OD ²⁾	a1	a2	b1	b2	b1	b2	b1	b2	b1	b2
	FF	1		1		1		1		1	
	FS	1		1		1		1		1	
	FC	0		1		0		0		0	
	IL	1		1		1		1		1	
	ST	0		0		0		0		0	
Field still picture	OT	a1	a2	b1	b2	b1	b2	b1	b2	b1	b2
	OD	a1	a2	b1	b2	b1	b2	b1	b2	b1	b2
	FF	1		1		1		1		1	
	FS	1		1		1		1		1	
	FC	0		1		0		0		0	
	IL	0		0		0		0		0	
	ST	0		0		0		0		0	

1) OT output order to the TV screen
2) OD output order to the digital interface

Reproducing frames from tape		indefinite	indefinite	indefinite	indefinite	indefinite
Reproducing fields from tape		indefinite	indefinite	indefinite	indefinite	indefinite
Interlaced motion picture	OT ¹⁾	indefinite	indefinite	indefinite	indefinite	indefinite
	OD ²⁾	indefinite	indefinite	indefinite	indefinite	indefinite
	FF	1	1	1	1	1
	FS	1	1	1	1	1
	FC	1	1	1	1	1
	IL	1	1	1	1	1
	ST	1	1	1	1	1
Non-interlaced motion picture	OT	indefinite	indefinite	indefinite	indefinite	indefinite
	OD	indefinite	indefinite	indefinite	indefinite	indefinite
	FF	1	1	1	1	1
	FS	1	1	1	1	1
	FC	1	1	1	1	1
	IL	1	1	1	1	1
	ST	1	1	1	1	1
Frame still picture	OT	indefinite	indefinite	indefinite	indefinite	indefinite
	OD	indefinite	indefinite	indefinite	indefinite	indefinite
	FF	1	1	1	1	1
	FS	1	1	1	1	1
	FC	1	1	1	1	1
	IL	1	1	1	1	1
	ST	1	1	1	1	1
Field still picture	OT	indefinite	indefinite	indefinite	indefinite	indefinite
	OD	indefinite	indefinite	indefinite	indefinite	indefinite
	FF	1	1	1	1	1
	FS	1	1	1	1	1
	FC	1	1	1	1	1
	IL	1	1	1	1	1
	ST	1	1	1	1	1

1) OT output order to the TV screen
2) OD output order to the digital interface

VAUX 2

9.3 REC DATE (Recording date)

	MSB				LSB			
PC 0	0	1	1	0	0	0	1	0
PC 1	DS	TM	TENS of TIME ZONE		UNITS of TIME ZONE			
PC 2	1	1	TENS of DAY		UNITS of DAY			
PC 3	WEEK			TNMN	UNITS of MONTH			
PC 4	TENS of YEAR				UNITS of YEAR			

This pack should be recorded in the VAUX main area. The date when video data are recorded is stored in this pack.

DS: Daylight saving time

0 = Daylight saving time

1 = Normal

TM: Thirty minutes flag

Thirty minutes unit of the time differential from GMT

0 = 30 min

1 = 0 min

TIME ZONE:

00 to 23

3Fh = No information

Example

For Tokyo

TIME ZONE = 001001b

PC1 = 11001001b

GMT plus 9:00

For New York with daylight saving time

TIME ZONE = 011001b

PC1 = 01011001b

GMT plus 19:00

For New Delhi where 30 min unit of the time differential from GMT is adopted.

TIME ZONE = 000101b

PC1 = 10000101b

GMT plus 5:30

where GMT: Greenwich Mean Time

DAY:

01 to 31

3Fh = No information

WEEK:

0 = Sunday

4 = Thursday

1 = Monday

5 = Friday

2 = Tuesday

6 = Saturday

3 = Wednesday

7 = No information

MONTH:

01 to 12 = January to December

1Fh = No information

TNMN: Tens of month

YEAR: Last two figures of year

00 to 99

FFh = No information

VAUX 3

9.4 REC TIME

This pack should be recorded in the VAUX main area.

The time when video data are recorded is stored based on the SMPTE/EBU time code format.

For not recording VAUX BINARY pack

	MSB				LSB			
PC 0	0	1	1	0	0	0	1	1
PC 1	1	1	TENS of FRAMES		UNITS of FRAMES			
PC 2	1	TENS of SECONDS		UNITS of SECONDS				
PC 3	1	TENS of MINUTES		UNITS of MINUTES				
PC 4	1	1	TENS of HOURS		UNITS of HOURS			

Consumer digital VCR shall adopt the drop frame sequence.

If FRAME is not used, FRAME shall be 3Fh.

For recording VAUX BINARY pack

	MSB								LSB	
PC 0	0	1	1	0	0	0	1	1		
PC 1	S2	S1	TENS of FRAMES			UNITS of FRAMES				
PC 2	S3	TENS of SECONDS			UNITS of SECONDS					
PC 3	S4	TENS of MINUTES			UNITS of MINUTES					
PC 4	S6	S5	TENS of HOURS			UNITS of HOURS				

S1 to S6 flags shall be recorded based on SMPTE/EBU format.

Bit number	S1	S2	S3	S4	S5	S6
VITC	14	15	35	55	74	75
LTC	10	11	27	43	58	59

VITC: vertical interval time code
LTC: linear time code

VAUX 4

9.5 BINARY GROUP

	MSB	LSB
PC 0	0 1 1 0 0 1 0 0	
PC 1	BINARY GROUP 2	BINARY GROUP 1
PC 2	BINARY GROUP 4	BINARY GROUP 3
PC 3	BINARY GROUP 6	BINARY GROUP 5
PC 4	BINARY GROUP 8	BINARY GROUP 7

This pack may be recorded in the VAUX main area.

If this pack is used, S1 to S6 flags in VAUX REC TIME pack shall be set based on the SMPTE/EBU time code format.

If this pack is not used, NO INFO pack shall be recorded.

VAUX 5

9.6 CLOSED CAPTION

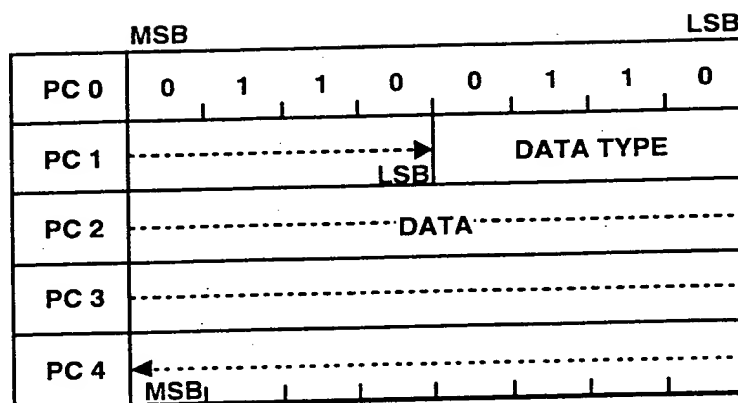
	MSB	LSB
PC 0	0 1 1 0 0 1 0 1	
PC 1	MSB 1st FIELD Line 21 1st BYTE	LSB
PC 2	MSB 1st FIELD Line 21 2nd BYTE	LSB
PC 3	MSB 2nd FIELD Line 21 1st BYTE	LSB
PC 4	MSB 2nd FIELD Line 21 2nd BYTE	LSB

This pack should be recorded in the VAUX main area.

Closed caption data should be stored in VAUX CLOSED CAPTION pack without change. The data shall be stored from next bit of start bits as a LSB. If the data which concern VAUX SOURCE, VAUX SOURCE CONTROL, AAUX SOURCE and AAUX SOURCE CONTROL packs, such as DISP, CLF and AUDIO MODE, are transmitted, the contents should be set in those four packs to avoid the inconsistency between VAUX CLOSED CAPTION pack and those four packs. If there exists information on audience restrictions, SS in SOURCE CONTROL packs of VAUX and AAUX shall be set to 10b. If VAUX CLOSED CAPTION packs have been recorded on tape, closed caption signals should be reconstructed and added to line 21 in each field of the vertical blanking period.

More details are given in 9.5 of Part 2.

9.7 TR (Transparent)



This pack should be recorded in the VAUX main area.

In addition to VAUX CLOSED CAPTION pack, VAUX TR pack is prepared for preserving digital data such as Video ID, WSS (wide screen signalling) and EDTV-2 ID without change. If these signals are transmitted in the vertical blanking period, VAUX TR pack should be recorded. If the data which concern VAUX SOURCE, VAUX SOURCE CONTROL, AAUX SOURCE and AAUX SOURCE CONTROL packs, such as DISP, CLF and AUDIO MODE, are transmitted, the contents should be set in those four packs to avoid inconsistency between VAUX TR pack and those four packs. If there is no data for the DATA area, all "1" data shall be recorded. If VAUX TR packs have been recorded on tape, the signals of DATA TYPE should be reconstructed and added in the appropriate lines of the vertical blanking period.

More details are given in 9.5 of Part 2.

DATA TYPE:

- 0 = Video ID
- 1 = WSS
- 2 = EDTV-2 ID in 22 line
- 3 = EDTV-2 ID in 285 line
- Fh = No information
- Others = Reserved

For recording Video ID data

Video ID data of one horizontal line consists of 20 bits. The data shall be stored from the side of horizontal sync as an LSB. All 20 bits of data shall be stored in the VAUX TR pack.

For recording WSS data

WSS data of one horizontal line consists of 14 bits. The data shall be stored from next bit of start bits as an LSB. All 14 bits of data shall be stored in the VAUX TR pack.

For recording EDTV-2 ID data

EDTV-2 ID data of one horizontal line consists of 27 bits. The data shall be stored from the side of horizontal sync as an LSB. 24 bits of the data except the last 3 bits for discriminator shall be stored in the VAUX TR pack.

9.8 TELETEXT

	MSB								LSB
PC 0	0	1	1	0	0	1	1	1	
PC 1									
PC 2									
PC 3	TELETEXT DATA								
PC 4									

This pack may be recorded in the VAUX common optional area.

Step 1: Gathering teletext data in one horizontal line

Following a teletext ID shown below, teletext data in one horizontal line shall be gathered and reconstructed in the form of bytes from the next bit of teletext framing code as an LSB to the end of teletext signal in order.

Step 2: Packing teletext data in TELETEXT packs

The TELETEXT IDs and TELETEXT data in one video frame shall be gathered in order and packed in the TELETEXT packs. If there exists the remains in the last TELETEXT pack, the data of FFh which is indicative of no information shall be filled.

Step 3: Recording teletext data in VAUX common optional area

The queue of teletext recording packs consists of a VAUX TEXT HEADER pack, a TELETEXT INFO pack, if needed, and TELETEXT packs. TEXT TYPE in VAUX TEXT HEADER pack shall be set to Ah. This queue should be recorded multiple times in one video frame. In the final TELETEXT pack in one video frame, the terminate code shall be recorded.

Teletext ID:

Teletext ID consists of System ID, Odd / Even and Line ID.

System ID	O/E	Line ID
b7 b6 b5	b4	b3 b2 b1 b0

System ID:

- 00b = Japanese teletext system (the bulletin number 803 of the Ministry of Postal Service in Japan – October 1985, teletext type D in ITU-R Recommendation 653)
- 01b = NABTS teletext system (teletext type C in ITU-R Recommendation 653)
- 10b = Reserved
- 11b = UK teletext system (EBU SPB492 – December 1992, teletext type B in ITU-R Recommendation 653)

O/E: Odd / Even

0 = Odd field or first field

1 = Even field or second field

Line ID: Line number ID

For 525-60 system

0 to 0Ch = Actual line number

0Dh to 1Eh = Reserved

1Fh = Terminate code

For O/E = 0, Actual line number = 10 + Line ID

For O/E = 1, Actual line number = 272 + Line ID

For 625-50 system

0 to 11h = Actual line number

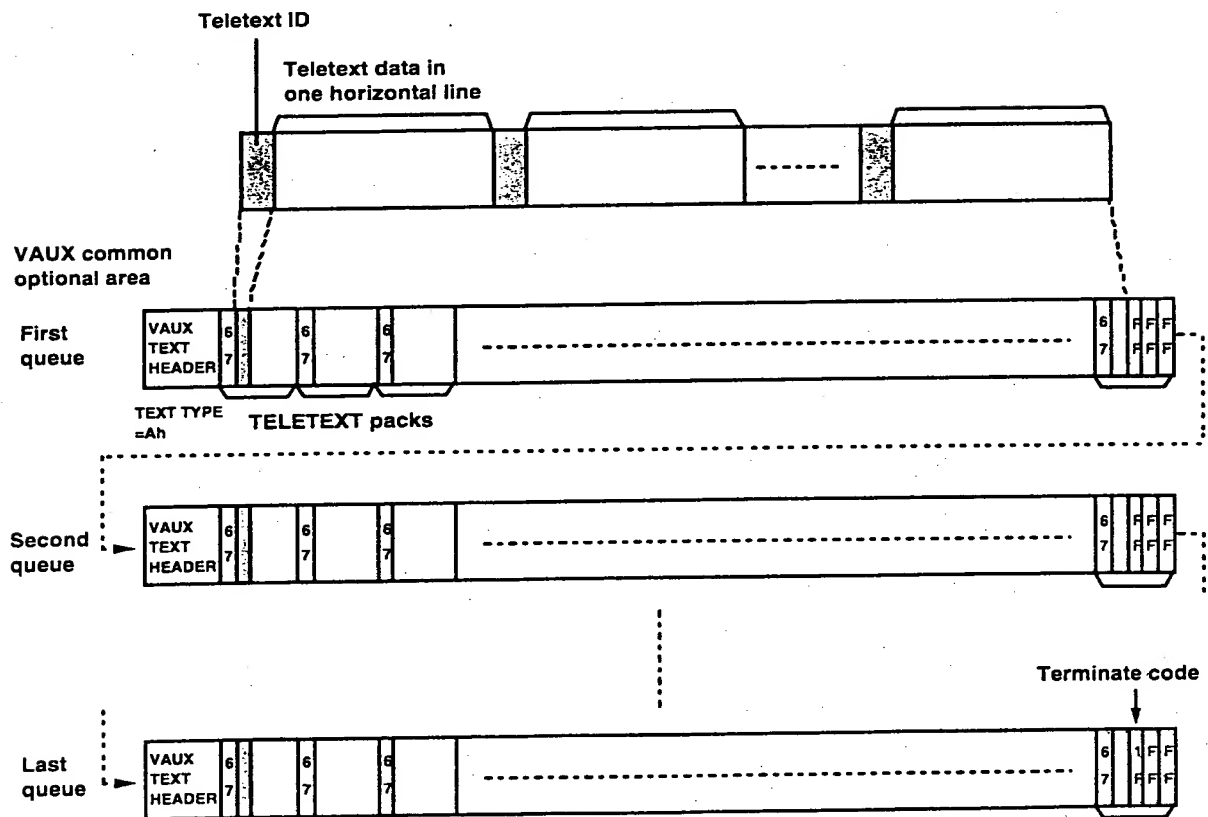
12h to 1Eh = Reserved

1Fh = Terminate code

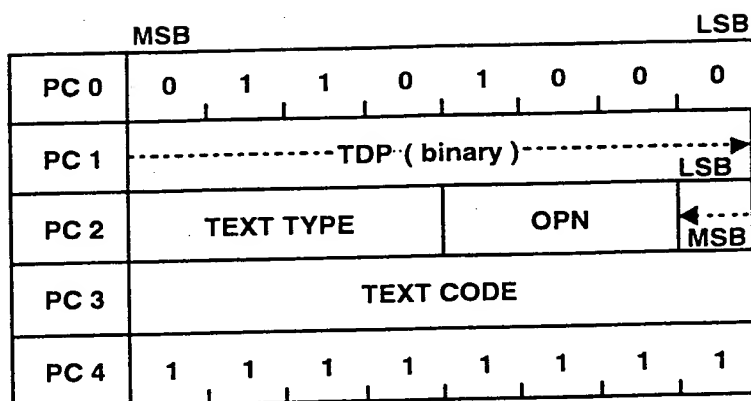
For O/E = 0, Actual line number = 6 + Line ID

For O/E = 1, Actual line number = 318 + Line ID

Procedure for recording teletext data



9.9 TEXT HEADER



This pack may be recorded or written in the common optional areas.

TDP: Total number of text data (see Figure 55 of Part 2)

For tape, total number of TEXT packs which follow this pack

For MIC, total number of text data bytes which follow PC3

TEXT TYPE:

0 = Name	7 = Subtitle	Dh = Two byte coded font
1 = Memo	8 = Outline	Eh = Graphic
2 = Station	9 = Full screen	Fh = No information
3 = Model	Ah = Teletext header	Others = Reserved
6 = Operator	Ch = One byte coded font	

OPN: Option number

OPN is the option number of UK teletext. More details are given in teletext specification (EBU SPB 492 – December 1992).

If OPN is not used, OPN shall be 111b.

TEXT CODE:

TEXT CODE designates the character set. The details are described in CONTROL TEXT HEADER pack.

VAUX 9

9.10 TEXT

	MSB								LSB
PC 0	0	1	1	0	1	0	0	1	
PC 1									
PC 2									
PC 3									
PC 4									

This pack may be recorded in the common optional areas on tape.

This pack contains font data, graphic data, text data according to TEXT TYPE designated in VAUX TEXT HEADER pack.

9.11 VAUX START

MSB									LSB
PC 0	0	1	1	0	1	0	1	0	
PC 1	1	DF	TENS of FRAMES			UNITS of FRAMES			
PC 2	TENS of SECONDS				UNITS of SECONDS				
PC 3	TENS of MINUTES				UNITS of MINUTES				
PC 4	TENS of HOURS				UNITS of HOURS				

This pack may be recorded or written in the common optional areas except for the AAUX optional area.

This pack shows the tape position of starting to insert video data using title time code.

DF: Drop frame flag

0 = Drop frame mode

1 = Non drop frame mode

Drop frame sequence shall be based on SMPTE/EBU format.

For consumer digital VCR, DF shall be 0.

FRAMES:

For 525-60 or 1125-60 system

00 to 29

For 625-50 or 1250-50 system

00 to 24

SECONDS:

00 to 59

MINUTES:

00 to 59

HOURS:

00 to 23

9.12 VAUX START

	MSB	LSB
PC 0	0 1 1 0 1 0 1 1	
PC 1	-----▶	TT
PC 2	-----ABSOLUTE TRACK NO.-----	
PC 3	◀----- (binary) -----	
PC 4	TEXT	GENRE CATEGORY

This pack may be recorded or written in the common optional areas except for the AAUX optional area.

This pack shows the tape position of starting to insert video data using absolute track number.

ABSOLUTE TRACK NO.:

Absolute track number which shows the tape position of starting to insert video data

TT: Temporary true

This flag is valid only for MIC.

0 = This event data in MIC does not always exist on tape

1 = This event data in MIC exists on tape certainly

For subcode, AAUX and VAUX, TT shall be 1.

TEXT:

This flag is valid only for MIC.

0 = Text information exists

1 = No text information exists

For subcode, AAUX and VAUX, TEXT shall be 1.

GENRE CATEGORY:

GENRE CATEGORY shows the category of the inserted video source.

The details are described in TIMER ACT DATE pack.

9.13 MARINE/MOUNTAIN

	MSB				LSB			
PC 0	0	1	1	0	1	1	0	0
PC 1	1/10 of TEMPERATURE				CF	CATEGO		0
PC 2	TENS of TEMPERATURE				UNITS of TEMPERATURE			
PC 3	UNITS of PRESSURE				THPR	0	NP	HDRT
PC 4	HUNDREDS of PRESSURE				TENS of PRESSURE			

This pack may be recorded or written in the common optional areas.

This pack contains the temperature and pressure data of the location where the recording was made.

CF: Centigrade/Fahrenheit

0 = Fahrenheit

1 = Centigrade

CATEGO: Category code

0 = Marine

1 = Mountain

Others = Reserved

NP: Negative/positive

NP shows the positive and negative sign of the temperature data.

0 = Negative

1 = Positive

PRESSURE:

0 000 hPa to 1 999 hPa

HPR: Thousands of pressure

TEMPERATURE:

000,0 to 199,9

HDRT: Hundreds of temperature

9.13 MARINE/MOUNTAIN (continued)

	MSB								LSB
PC 0	0	1	1	0	1	1	0	0	
PC 1	1/10 of TEMPERATURE				CF	CATEGO		0	
PC 2	TENS of TEMPERATURE				UNITS of TEMPERATURE				
PC 3	1/10 of ATM PRESSURE				HDPR	1	NP	HDRT	
PC 4	TENS of ATM PRESSURE				UNITS of ATM PRESSURE				

This pack may be recorded or written in the common optional areas.

This pack contains the temperature and pressure data of the location where the recording was made.

CF: Centigrade/Fahrenheit

0 = Fahrenheit

1 = Centigrade

CATEGO: Category code

0 = Marine

1 = Mountain

Others = Reserved

NP: Negative/positive

NP shows the positive and negative sign of the temperature data.

0 = Negative

1 = Positive

ATM PRESSURE:

000,0 atm to 199,9 atm

where atm = hPa / 1 013,25

HDPR: Hundreds of atm pressure

TEMPERATURE:

000,0 to 199,9

HDRT: Hundreds of temperature

9.13 MARINE/MOUNTAIN (concluded)

MSB					LSB			
PC 0	0	1	1	0	1	1	0	0
PC 1	1/10 of HEIGHT				FM	CATEGO		1
PC 2	TENS of HEIGHT				UNITS of HEIGHT			
PC 3	THOUSANDS of HEIGHT				HUNDREDS of HEIGHT			
PC 4	1	1	1	NP	TEN THOUSANDS of HEIGHT			

This pack may be recorded or written in the common optional areas.

This pack contains the height and depth data of the location where the recording was made.

FM: Feet/meter

0 = Feet

1 = Meter

CATEGO: Category code

0 = Marine

1 = Mountain

Others = Reserved

NP: Negative/positive

NP shows the positive and negative sign of the height data.

0 = Negative

1 = Positive

HEIGHT:

00 000,0 to 99 999,9

9.14 LONGITUDE/LATITUDE

	MSB								LSB
PC 0	0	1	1	0	1	1	0	1	
PC 1	0	TENS of SECOND				UNITS of SECOND			
PC 2	EW	TENS of MINUTE				UNITS of MINUTE			
PC 3	TENS of DEGREE				UNITS of DEGREE				
PC 4	1	1	1	1	1	1	1	1	HDRD

This pack may be recorded or written in the common optional areas.

This pack contains the longitude data of the location where the recording was made.

SECOND:

00 to 59

MINUTE:

00 to 59

DEGREE:

00 to 180

HDRD: Hundreds of degrees

Longitude data has a valid range of 0° 00'00" to 180° 00'00".

EW: East/West

0 = East

1 = West

9.14 LONGITUDE/LATITUDE (concluded)

	MSB								LSB
PC 0	0	1	1	0	1	1	0	1	
PC 1	1	TENS of SECOND				UNITS of SECOND			
PC 2	NS	TENS of MINUTE				UNITS of MINUTE			
PC 3	TENS of DEGREE				UNITS of DEGREE				
PC 4	1	1	1	1	1	1	1	1	

This pack may be recorded or written in the common optional areas.

This pack contains the latitude data of the location where the recording was made.

SECOND:

00 to 59

MINUTE:

00 to 59

DEGREE:

00 to 90

Latitude data has a valid range of 0° 00'00" to 90° 00'00".

NS: North/South

0 = North

1 = South

9.15 VAUX END

	MSB								LSB
PC 0	0	1	1	0	1	1	1	0	
PC 1	1	DF	TENS of FRAMES			UNITS of FRAMES			
PC 2	TENS of SECONDS				UNITS of SECONDS				
PC 3	TENS of MINUTES				UNITS of MINUTES				
PC 4	TENS of HOURS				UNITS of HOURS				

This pack may be recorded or written in the common optional areas except for the AAUX optional area.

This pack shows the tape position of ending to insert video data using title time code.

DF: Drop frame flag

0 = Drop frame mode

1 = Non drop frame mode

Drop frame sequence shall be based on SMPTE/EBU format.

For consumer digital VCR, DF shall be 0.

FRAMES:

For 525-60 or 1125-60 system

00 to 29

For 625-50 or 1250-50 system

00 to 24

SECONDS:

00 to 59

MINUTES:

00 to 59

HOURS:

00 to 23

9.16 VAUX END

	MSB	LSB
PC 0	0 1 1 0 1 1 1	1
PC 1	-----> LSB	BF
PC 2	-----ABSOLUTE TRACK NO.-----	
PC 3	<----- (binary) -----	
PC 4	1 1 1	TNT 1 1

This pack may be recorded or written in the common optional areas except for the AAUX optional area.

This pack shows the tape position of ending to insert video data using absolute track number.

ABSOLUTE TRACK NO.:

Absolute track number which shows the end tape position of video insert

BF: Blank flag

0 = Discontinuity exists before this absolute track number.

1 = Discontinuity does not exist before this absolute track number.

TNT: Total number of text events

TNT is valid only for MIC.

TNT shows the total number of text events related to this VAUX event.

0 to 6 7 = No information

For subcode, AAUX and VAUX, TNT shall be 111b.

JTSB-508-US (3)

INTERNET-DRAFT
draft-dasl-protocol-00.txt

Saveen Reddy, Microsoft
Dale Lowry, Novell
Surendra Reddy, Oracle
Rick Henderson, Netscape
Jim Davis, CourseNet
Alan Babich, FileNet

Expires December 3, 1999

June 3, 1999

DAV Searching & Locating

Status of this Memo

This document is an Internet draft. Internet drafts are working documents of the Internet Engineering Task Force (IETF), its areas and its working groups. Note that other groups may also distribute working information as Internet drafts.

Internet Drafts are draft documents valid for a maximum of six months and can be updated, replaced or obsoleted by other documents at any time. It is inappropriate to use Internet drafts as reference material or to cite them as other than as "work in progress".

To view the entire list of current Internet-Drafts, please check the "lid-abstracts.txt" listing contained in the Internet-Drafts Shadow Directories on ftp.is.co.za (Africa), ftp.nordu.net (Northern Europe), ftp.nis.garr.it (Southern Europe), munnari.oz.au (Pacific Rim), ftp.ietf.org (US East Coast), or ftp.isi.edu (US West Coast).

Distribution of this document is unlimited. Please send comments to the mailing list at <www-webdav-dasl@w3.org>, which may be joined by sending a message with subject "subscribe" to <www-webdav-dasl-request@w3.org>.

Discussions of the list are archived at
<URL:<http://www.w3.org/pub/WWW/Archives/Public/www-webdav-dasl>>.

Abstract

This document specifies a set of methods, headers, and content-types composing DASL, an application of the HTTP/1.1 protocol to efficiently search for DAV resources based upon a set of client-supplied criteria.

1. Introduction

1.1 DASL

This document defines DAV Searching & Locating (DASL), an application of HTTP/1.1 forming a lightweight search protocol to transport queries and result sets and allows clients to make use of server-side search facilities. [DASLREQ] describes the motivation for DASL.

DASL will minimize the complexity of clients so as to facilitate widespread deployment of applications capable of utilizing the DASL search mechanisms.

DASL consists of:

- the SEARCH method,
- the DASL response header,
- the DAV:searchrequest XML element,
- the DAV:querschema property,
- the DAV:basicsearch XML element and query grammar, and
- the DAV:basicsearchschema XML element.

1.2 Relationship to DAV

DASL relies on the resource and property model defined by [WebDAV]. DASL does not alter this model. Instead, DASL allows clients to access DAV-modeled resources through server-side search.

1.3 Terms

This draft uses the terms defined in [RFC2068], [WebDAV], and [DASLREQ].

1.4 Notational Conventions

The augmented BNF used by this document to describe protocol elements is exactly the same as the one described in Section 2.1 of [RFC2068]. Because this augmented BNF uses the basic production rules provided in Section 2.2 of [RFC2068], those rules apply to this document as well.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

1.5 An Overview of DASL at Work

One can express the basic usage of DASL in the following steps:

- The client constructs a query using the DAV:basicsearch grammar.
- The client invokes the SEARCH method on a resource that will perform the search (the search arbiter) and includes a text/xml request entity that contains the query.
- The search arbiter performs the query.
- The search arbiter sends the results of the query back to the client in the response. The server MUST send a text/xml entity that matches the [WebDAV] PROPFIND response.

2. The SEARCH Method

2.1 Overview

The client invokes the SEARCH method to initiate a server-side search. The body of the request defines the query. The server MUST emit text/xml entity matching the [WebDAV] PROPFIND response.

The SEARCH method plays the role of transport mechanism for the query and the result set. It

does not define the semantics of the query. The type of the query defines the semantics.

2.2 The Request

The client invokes the SEARCH method on the resource named by the Request-URI.

2.2.1 The Request-URI

The Request-URI identifies the search arbiter.

The SEARCH method defines no relationship between the arbiter and the scope of the search, rather the particular query grammar used in the query defines the relationship. For example, the FOO query grammar may force the request-URI to correspond exactly to the search scope.

2.2.2 The Request Body

The server **MUST** process a text/xml or application/xml request body, and **MAY** process request bodies in other formats. See [RFC 2376] for guidance on packaging XML in requests.

If the client sends a text/xml or application/xml body, it **MUST** include the `DAV:searchrequest` XML element. The `DAV:searchrequest` XML element identifies the query grammar, defines the criteria, the result record, and any other details needed to perform the search.

2.3 The `DAV:searchrequest` XML Element

```
<!ELEMENT searchrequest ANY >
```

The `DAV:searchrequest` XML element contains a single XML element that defines the query. The name of the query element defines the type of the query. The value of that element defines the query itself.

2.4 The Successful 207 (Multistatus) Response

If the server returns 207 (Multistatus), then the search proceeded successfully and the response **MUST** match that of a PROPFIND.

There **MUST** be one `DAV:response` for each resource that matched the search criteria. For each such response, the `DAV:href` element contains the URI of the resource, and the response **MUST** include a `DAV:propstat` element.

In addition, the server **MAY** include `DAV:response` items in the reply where the `DAV:href` element contains a URI that is not a matching resource, e.g. that of a scope or the query arbiter. Each such response item **MUST NOT** contain a `DAV:propstat` element, and **MUST** contain a `DAV:status`. It **SHOULD** contain a `DAV:responsedescription`.

2.4.1 Extending the PROPFIND Response

A response **MAY** include more information than PROPFIND defines so long as the extra information does not invalidate the PROPFIND response. Query grammars **SHOULD** define how the response matches the PROPFIND response.

2.4.1 Example: A Simple Request and Response

This example demonstrates the request and response framework. The following XML document shows a simple (hypothetical) natural language query. The name of the query element is FOO:natural-language-query, thus the type of the query is FOO:natural-language-query. The actual query is "Find the locations of good Thai restaurants in Los Angeles". For this hypothetical query, the arbiter returns two properties for each selected resource.

```
SEARCH / HTTP/1.1
Host: ryu.com
Content-Type: text/xml
Connection: Close
Content-Length: 243

<?xml version="1.0"?>
<D:searchrequest xmlns:D = "DAV:" xmlns:F = "FOO:">
  <F:natural-language-query>
    Find the locations of good Thai restaurants in Los Angeles
  </F:natural-language-query>
</D:searchrequest>
```

>> Response

```
HTTP/1.1 207 Multi-Status
Content-Type: text/xml
Content-Length: 333

<?xml version="1.0"?>
<D:multistatus xmlns:D="DAV:" xmlns:F="FOO:"
  xmlns:R="http://ryu.com/propschema">
  <D:response>
    <D:href>http://siamiam.com/</D:href>
    <D:propstat>
      <D:prop>
        <R:location>259 W. Hollywood</R:location>
        <R:rating><R:stars>4</R:stars></R:rating>
      </D:prop>
    </D:propstat>
  </D:response>
</D:multistatus>
```

2.5 Unsuccessful Responses

If an error occurred that prevented execution of the query, the server **MUST** indicate the failure with the appropriate status code and **SHOULD** include a DAV:multistatus element to point out errors associated with scopes.

400 Bad Request. The query could not be executed. The request may be malformed (not valid XML for example). Additionally, this can be used for invalid scopes and search redirections.

422 Unprocessable entity. The query could not be executed. If a text/xml request entity was provided, then it may have been valid (well-formed) but may have contained an unsupported or unimplemented query operator.

507 (Insufficient Storage). The query produced more results than the server was willing to transmit. Partial results have been transmitted. The server **MUST** send a body that matches that for 207, except that there **MAY** exist resources that matched the search criteria for which no corresponding DAV:response exists in the reply.

2.5.1 Example: Result Set Truncation

A server MAY limit the number of resources in a reply, for example to limit the amount of resources expended in processing a query. If it does so, the reply MUST use status code 507. It SHOULD include the partial results.

When a result set is truncated, there may be many more resources that satisfy the search criteria but that were not examined.

If partial results are included and the client requested an ordered result set in the original request, then any partial results that are returned MUST be ordered as the client directed.

Note that the partial results returned MAY be any subset of the result set that would have satisfied the original query.

```
SEARCH / HTTP/1.1
Host: gdr.com
Content-Type: text/xml
Connection: Close
Content-Length: xxxxx
```

```
<?xml version="1.0"?>
<D:searchrequest xmlns:D="DAV:">
  <D:basicsearch>
    ... the query goes here ...
  </D:basicsearch>
</D:searchrequest>
```

>> Response

```
HTTP/1.1 507 Insufficient Storage
Content-Type: text/xml
Content-Length: 738
```

```
<?xml version="1.0"?>
<D:multistatus xmlns:D="DAV:">
  <D:response>
    <D:href>http://www.gdr.com/sounds/unbrokenchain.au</D:href>
    <D:propstat>
      <D:prop/>
      <D:status>HTTP/1.1 200 OK</D:status>
    </D:propstat>
  </D:response>
  <D:response>
    <D:href>http://tech.mit.edu/archive96/photos/Lesh1.jpg</D:href>
    <D:propstat>
      <D:prop/>
      <D:status>HTTP/1.1 200 OK</D:status>
    <D:/propstat>
  </D:response>
  <D:response>
    <D:href>http://gdr.com</D:href>
    <D:status>HTTP/1.1 507 Insufficient Storage</D:status>
    <D:responsedescription>
      Only first two matching records were returned
    </D:responsedescription>
  </D:response>
</D:multistatus>
```

2.6 Invalid Scopes & Search Redirections

2.6.1 Indicating an Invalid Scope

A client may submit a scope that the arbiter may be unable to query. The inability to query may be due to network failure, administrative policy, security, etc. This raises the condition described as an "invalid scope".

To indicate an invalid scope, the server **MUST** respond with a 400 (Bad Request).

The response includes a text/xml body with a DAV:multistatus element. Each DAV:resource in the DAV:multistatus identifies a scope. To indicate that this scope is the source of the error, the server **MUST** include the DAV:scopeerror element.

2.6.2 Example of an Invalid Scope

```
HTTP/1.1 400 Bad-Request
Content-Type: text/xml
Content-Length: xxxxxx

<?xml version="1.0" ?>

<d:multistatus xmlns:d="DAV:">
  <d:response>
    <d:href>http://www.foo.com/X</d:href>
    <d:status>HTTP/1.1 404 Object Not Found</d:status>
    <d:scopeerror/>
  </d:response>
</d:multistatus>
```

2.6.3 Redirections

As described above, a server can indicate only that the scope is invalid. Some search arbiters may be able to indicate that other search arbiters exist for that scope.

In this case, the server **MUST**:

- (1) include the DAV:scopeerror element
- (2) include the DAV:status element for that scope. The value of this element **MUST** be a 303 (See Other) response.
- (3) include the DAV:redirectarbiter element for each arbiter the client should use for the redirect. The value of this element is the URI of the arbiter to use. Multiple DAV:redirectarbiter elements are allowed.

2.6.4 Example of a Search Redirection

```
HTTP/1.1 400 Bad-Request
Content-Type: text/xml
Content-Length: xxxxxx

<?xml version="1.0" ?>
<?xml:namespace ns="DAV:" prefix="d" ?>

<d:multistatus>
  <d:response>
    <d:href>http://www.foo.com/X</d:href>
    <d:status>HTTP/1.1 303 See Other</d:status>
```

```

    <d:scopeerror/>
    <d:redirectarbiter>http://bar.com/B</d:redirectarbiter>
    <d:redirectarbiter>http://baz.com/B</d:redirectarbiter>
  </d:response>
</d:multistatus>

```

2.6.5 Syntax for DAV:scopeerror

```
<!ELEMENT scopeerror EMPTY>
```

2.6.6 Syntax for DAV:redirectarbiter

```
<!ELEMENT redirectarbiter (#PCDATA)>
```

The contents must be a URL.

3. Discovery of Supported Query Grammars

Servers **MUST** support discovery of the query grammars supported by a search arbiter resource.

Clients can determine which query grammars are supported by an arbiter by invoking **OPTIONS** on the search arbiter. If the resource supports **SEARCH**, then the **DASL** response header will appear in the response. The **DASL** response header lists the supported grammars.

3.1 The OPTIONS Method

The **OPTIONS** method allows the client to discover if a resource supports the **SEARCH** method and to determine the list of search grammars supported for that resource.

The client issues the **OPTIONS** method against a resource named by the Request-URI. This is a normal invocation of **OPTIONS** defined in [RFC2068].

If a resource supports the **SEARCH** method, then the server **MUST** list **SEARCH** in the **OPTIONS** response as defined by [RFC2068].

DASL servers **MUST** include the **DASL** header in the **OPTIONS** response. This header identifies the search grammars supported by that resource.

3.2 The DASL Response Header

```

DASLHeader = "DASL" ":" Coded-URL-List
Coded-URL-List : Coded-URL [ "," Coded-URL-List ]
Coded-URL ; defined in section 9.4 of [WEBDAV]

```

The **DASL** response header indicates server support for a query grammar in the **OPTIONS** method. The value is a URI that indicates the type of grammar. This header **MAY** be repeated.

For example:

```

DASL: <http://foo.bar.com/syntax1>
DASL: <http://akuma.com/syntax2>
DASL: <FOO:natural-language-query>

```

3.3 Example: Grammar Discovery

This example shows that the server supports search on the /somefolder resource with the query grammars: DAV:basicsearch, http://foo.bar.com/syntax1 and http://akuma.com/syntax2. Note that every server **MUST** support DAV:basicsearch.

>> Request

```
OPTIONS /somefolder HTTP/1.1
Connection: Close
Host: ryu.com
```

>> Response

```
HTTP/1.1 200 OK
Date: Tue, 20 Jan 1998 20:52:29 GMT
Connection: close
Accept-Ranges: none
Allow: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE, COPY, MOVE, MKCOL, PROPFIND
Public: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE, COPY, MOVE, MKCOL, PROPFIND
DASL: <DAV:basicsearch>
DASL: <http://foo.bar.com/syntax1>
DASL: <http://akuma.com/syntax2>
```

4. Query Schema Discovery: QSD

Servers **MAY** support the discovery of the schema for a query grammar.

The DASL response header provides means for clients to discover the set of query grammars supported by a resource. This alone is not sufficient information for a client to generate a query. For example, the DAV:basicsearch grammar defines a set of queries consisting of a set of operators applied to a set of properties and values, but the grammar itself does not specify which properties may be used in the query. QSD for the DAV:basicsearch grammar allows a client to discover the set of properties that are searchable, selectable, and sortable. Moreover, although the DAV:basicsearch grammar defines a minimal set of operators, it is possible that a resource might support additional operators in a query. For example, a resource might support an optional operator that can be used to express content-based queries in a proprietary syntax. QSD allows a client to discover these operators and their syntax. The set of discoverable quantities will differ from grammar to grammar, but each grammar can define a means for a client to discover what can be discovered.

In general, the schema for a given query grammar depends on both the resource (the arbiter) and the scope. A given resource might have access to one set of properties for one potential scope, and another set for a different scope. For example, consider a server able to search two distinct collections, one holding cooking recipes, the other design documents for nuclear weapons. While both collections might support properties such as author, title, and date, the first might also define properties such as calories and preparation time, while the second defined properties such as yield and applicable patents. Two distinct arbiters indexing the same collection might also have access to different properties. For example, the recipe collection mentioned above might also be indexed by a value-added server that also stored the names of chefs who had tested the recipe. Note also that the available query schema might also depend on other factors, such as the identity of the principal conducting the search, but these factors are not exposed in this protocol.

Each query grammar supported by DASL defines its own syntax for expressing the possible query schema. A client retrieves the schema for a given query grammar on an arbiter resource with a given scope by invoking the SEARCH method on that arbiter, with that grammar and scope, with a query whose `DAV:select` element includes the `DAV:queryschema` property. This property is defined only in the context of such a search, a server SHOULD not treat it as defined in the context of a PROPFIND on the scope. The content of this property is an XML element whose name and syntax depend upon the grammar, and whose value may (and likely will) vary depending upon the grammar, arbiter, and scope.

The query schema for `DAV:basicsearch` is defined in section 5.19.

4.1 The `DAV:queryschema` Property

<!ELEMENT queryschema ANY >

4.1.1 Example of query schema discovery

In this example, the arbiter is `recipes.com`, the grammar is `DAV:basicsearch`, the scope is also `recipes.com`.

```
SEARCH / HTTP/1.1
Host: recipes.com
Content-Type: application/xml
Connection: Close
Content-Length: xxx

<?xml version="1.0"?>
<D:searchrequest xmlns:D="DAV:" >
  <D:basicsearch>
    <D:select>
      <D:queryschema/>
    </D:select>
    <D:from><D:scope><D:href>http://recipes.com</d:href></D:scope></D:from>
  </D:basicsearch>
</D:searchrequest>
```

Response:

```
HTTP/1.1 207 Multistatus
Content-Type: application/xml
Content-Length: xxx

<?xml version="1.0"?>
<D:multistatus xmlns:D="DAV:">
  <D:response>
    <D:href>http://recipes.com</D:href>
    <D:propstat>
      <D:prop>
        <D:querygrammar>
          <D:basicsearchschema>
            See section 5.19.9 for actual contents
          </D:basicsearchschema>
        </D:querygrammar>
      </D:prop>
    <D:status>HTTP/1.1 200 Okay</D:status>
  </D:response>
</D:multistatus>
```

5 The DAV:basicsearch Grammar

5.1 Introduction

DAV:basicsearch uses an extensible XML syntax that allows clients to express search requests that are generally useful for WebDAV scenarios. DASL-extended servers **MUST** accept this grammar, and **MAY** accept others grammars.

DAV:basicsearch has several components:

1. DAV:select provides the result record definition.
2. DAV:from defines the scope.
3. DAV:where defines the criteria.
4. DAV:orderby defines the sort order of the result set.
5. DAV:limit provides constraints on the query as a whole.

5.2 The DAV:basicsearch DTD

```
<!ELEMENT basicsearch      (select, from, where?, orderby?, limit?) >
<!ELEMENT select           (allprop | prop) >
<!ELEMENT from             (scope) >
<!ELEMENT scope            (href, depth?) >

<!ENTITY %comp_ops        "eq | lt | gt| lte | gte">
<!ENTITY %log_ops         "and | or | not">
<!ENTITY %special_ops     "isdefined">
<!ENTITY %string_ops      "like">
<!ENTITY %content_ops     "contains">

<!ENTITY %all_ops         "%comp_ops; | %log_ops; | %special_ops; | %string_ops; | %content_ops;">

<!ELEMENT where ( %all_ops; ) >
<!ELEMENT and   ( ( %all_ops; ) + ) >
<!ELEMENT or    ( ( %all_ops; ) + ) >
<!ELEMENT not   ( %all_ops; ) >

<!ELEMENT lt    ( prop , literal ) >
<!ATTLIST lt    casesensitive (1|0) "1" >

<!ELEMENT lte   ( prop , literal ) >
<!ATTLIST lte   casesensitive (1|0) "1" >

<!ELEMENT gt    ( prop , literal ) >
<!ATTLIST gt    casesensitive (1|0) "1" >

<!ELEMENT gte   ( prop , literal ) >
<!ATTLIST gte   casesensitive (1|0) "1" >

<!ELEMENT eq    ( prop , literal ) >
<!ATTLIST eq    casesensitive (1|0) "1" >

<!ELEMENT literal      (#PCDATA)>
<!ATTLIST literal      xml:space      (default|preserve) preserve >
```

```

<!ELEMENT isdefined      (prop) >
<!ELEMENT like (prop, literal) >
<!ELEMENT contains      (#PCDATA)>

<!ELEMENT orderby      (order+) >
<!ELEMENT order (prop, (ascending | descending)?)

<!ATTLIST order casesensitive (1|0) "1" >
<!ELEMENT ascending      EMPTY>

<!ELEMENT descending      EMPTY>

<!ELEMENT limit (nresults) >
<!ELEMENT nresults      (#PCDATA) >

```

5.2.1 Example Query

This query retrieves the content length values for all resources located under the server's "/container1/" URI namespace whose length exceeds 10000.

```

<d:searchrequest>
  <d:basicsearch>
    <d:select>
      <d:prop><d:getcontentlength/></d:prop>
    </d:select>
    <d:from>
      <d:scope>
        <d:href>/container1/</d:href>
        <d:depth>infinity</d:depth>
      </d:scope>
    </d:from>
    <d:where>
      <d:gt>
        <d:prop><d:getcontentlength/></d:prop>
        <d:literal>10000</d:literal>
      </d:gt>
    </d:where>
    <d:orderby>
      <d:order>
        <d:prop><d:getcontentlength/><d:prop>
        <d:ascending/>
      </d:order>
    </d:orderby>
  </d:basicsearch>
</d:searchrequest>

```

5.3 DAV:select

DAV:select defines the result record, which is a set of properties and values. This document defines two possible values: DAV:allprop and DAV:prop, both defined in [WebDAV].

If the value is DAV:allprop, the result record for a given resource includes all the properties for that resource.

If the value is DAV:prop, then the result record for a given resource includes only those properties named by the DAV:prop element. Each property named by the DAV:prop element must be referenced in the Multistatus response.

The rules governing the status codes for each property match those of the PROPFIND method

defined in [WebDAV].

5.4 DAV:from

DAV:from defines the query scope. This contains exactly one DAV:scope element. The scope element contains a mandatory DAV:href element and an optional DAV:depth element.

DAV:href indicates the URI for a collection to use as a scope.

When the scope is a collection, if DAV:depth is "0", the search includes only the collection. When it is "1", the search includes the (toplevel) members of the collection. When it is "infinity", the search includes all recursive members of the collection.

5.4.1 Relationship to the Request-URI

If the DAV:scope element is an absolute URI, the scope is exactly that URI.

If the DAV:scope element is a relative URI, the scope is taken to be relative to the request-URI.

5.4.2 Scope

A Scope can be an arbitrary URI.

Servers, of course, may support only particular scopes. This may include limitations for particular schemes such as "http:" or "ftp:" or certain URI namespaces.

If a scope is given that is not supported the server MUST respond with a 400 status code that includes a Multistatus error. A scope in the query appears as a resource in the response and must include an appropriate status code indicating its validity with respect to the search arbiter.

Example:

```
HTTP/1.1 400 Bad Request
Content-Type: text/xml
Content-Length: 428
```

```
<?xml version="1.0" ?>
<d:multistatus xmlns:D="DAV:" xmlns:F="FOO:" >
  <d:response>
    <d:href>http://www.foo.com/scopel</d:href>
    <d:status>HTTP/1.1 502 Bad Gateway</d:status>
  </d:response>
</d:multistatus>
```

This example shows the response if there is a scope error. The response provides a Multistatus with a status for the scope. In this case, the scope cannot be reached because the server cannot search another server (502).

5.5 DAV:where

DAV:where element defines the search condition for inclusion of resources in the result set. The value of this element is an XML element that defines a search operator that evaluates to one of the Boolean truth values TRUE, FALSE, or UNKNOWN. The search operator contained by DAV:where may itself contain and evaluate additional search operators as operands, which in turn

may contain and evaluate additional search operators as operands, etc. recursively.

5.5.1 Use of Three-Valued Logic in Queries

Each operator defined for use in the where clause that returns a Boolean value MUST evaluate to TRUE, FALSE, or UNKNOWN. The resource under scan is included as a member of the result set if and only if the search condition evaluates to TRUE.

Consult Appendix A for details on the application of three-valued logic in query expressions.

5.5.2 Handling Optional operators

If a query provides an operator that is not supported by the server, then the server MUST respond with a 422 (Unprocessable Entity) status code.

5.5.3 Treatment of NULL Values

If a SEARCH PROPFIND for a property value would yield a 404 or 403 response for that property, then that property is considered NULL.

NULL values are "less than" all other values in comparisons.

Empty strings (zero length strings) are not NULL values. An empty string is "less than" a string with length greater than zero.

The DAV:isdefined operator is defined to test if the value of a property is NULL.

5.5.4 Example: Testing for Equality

The example shows a single operator (DAV:eq) applied in the criteria.

```
<d:where>
  <d:eq>
    <d:prop> <d:getcontentlength/> </d:prop>
    <d:literal> 100 </d:literal>
  </d:eq>
</d:where>
```

5.5.5 Example: Relative Comparisons

The example shows a more complex operation involving several operators (DAV:and, DAV:eq, DAV:gt) applied in the criteria. This DAV:where expression matches those resources that are "image/gifs" over 4K in size.

```
<D:where>
  <D:and>
    <D:eq>
      <D:prop> <D:getcontenttype/> </D:prop>
      <D:literal> image/gif </D:literal>
    </D:eq>
    <D:gt>
      <D:prop> <D:getcontentlength/> </D:prop>
      <D:literal> 4096 </D:literal>
    </D:gt>
  </D:and>
```

```
</D:where>
```

5.6 DAV:orderby

The DAV:orderby element specifies the ordering of the result set. It contains one or more DAV:order elements, each of which specifies a comparison between two items in the result set. Informally, a comparison specifies a test that determines whether one resource appears before another in the result set. Comparisons are applied in the order they occur in the DAV:orderby element, earlier comparisons being more significant.

The comparisons defined here use only a single property from each resource, compared using the same ordering as the DAV:lt operator (ascending) or DAV:gt operator (descending). If neither direction is specified, the default is DAV:ascending.

In the context of the DAV:orderby element, null values are considered to collate before any actual (i.e., non null) value, including strings of zero length (as in ANSI standard SQL, [ANSISQL]).

5.6.1 Comparing Natural Language Strings.

Comparisons on strings take into account the language defined for that property. Clients MAY specify the language using the xml:lang attribute. If no language is specified either by the client or defined for that property by the server or if a comparison is performed on strings of two different languages, the results are undefined.

The DAV:casesensitive attribute may be used to indicate case-sensitivity for comparisons.

5.6.2 Example of Sorting

This sort orders first by last name of the author, and then by size, in descending order, so that the largest works appear first.

```
<d:orderby>
  <d:order>
    <d:prop><r:lastname/></d:prop>
    <d:ascending/>
  </d:order>
  <d:order>
    <d:prop><d:getcontentlength/></d:prop>
    <d:descending/>
  </d:order>
</d:orderby>
```

5.7 Boolean Operators: DAV:and, DAV:or, and DAV:not

The DAV:and operator performs a logical AND operation on the expressions it contains.

The DAV:or operator performs a logical OR operation on the values it contains.

The DAV:not operator performs a logical NOT operation on the values it contains.

5.8 DAV:eq

The DAV:eq operator provides simple equality matching on property values.

The `DAV:casesensitive` attribute may be used with this element.

5.9 `DAV:lt`, `DAV:lte`, `DAV:gt`, `DAV:gte`

The `DAV:lt`, `DAV:lte`, `DAV:gt`, and `DAV:gte` operators provide comparisons on property values, using less-than, less-than or equal, greater-than, and greater-than or equal respectively. The `DAV:casesensitive` attribute may be used with these elements.

5.10 `DAV:literal`

`DAV:literal` allows literal values to be placed in an expression.

Because white space in literal values is significant in comparisons, `DAV:literal` makes use of the `xml:space` attribute to identify this significance. The default value of this attribute for `DAV:literal` is `preserve`. Consult section 2.10 of [XML] for more information on the use of this attribute.

5.11 `DAV:isdefined`

The `DAV:isdefined` operator allows clients to determine whether a property is defined on a resource. The meaning of "defined on a resource" is found in section 5.5.3.

Example:

```
<d:isdefined>
  <d:prop><x:someprop/></d:prop>
</d:isdefined>
```

The `DAV:isdefined` operator is optional.

5.12 `DAV:like`

The `DAV:like` is an optional operator intended to give simple wildcard-based pattern matching ability to clients.

The operator takes two arguments.

The first argument is a `DAV:prop` element identifying a single property to evaluate.

The second argument is a `DAV:literal` element that gives the pattern matching string.

5.12.1 Syntax for the Literal Pattern

```
Pattern := [wildcard] 0*( text [wildcard] )
wildcard := exactlyone | zeroormore
text := 1*( <octet> | escapesequenece )
exactlyone := "?"
zeroormore := "*"
escapechar := "\"
escapesequenece := "\" ( exactlyone | zeroormore | escapechar )
```

The value for the literal is composed of wildcards separated by segments of text. Wildcards may

begin or end the literal. Wildcards may not be adjacent.

The "?" wildcard matches exactly one character.

The "%" wildcard matches zero or more characters

The "\" character is an escape sequence so that the literal can include "?" and "%". To include the "\" character in the pattern, the escape sequence "\\" is used..

5.12.2 Example of DAV:like

This example shows how a client might use DAV:like to identify those resources whose content type was a subtype of image.

```
<D:where>
  <D:like>
    <D:prop><D:getcontenttype/></D:prop>
    <D:literal>image%</D:literal>
  </D:like>
</D:where>
```

5.13 DAV:contains

The DAV:contains operator is an optional operator that provides content-based search capability. This operator implicitly searches against the text content of a resource, not against content of properties. The DAV:contains operator is intentionally not overly constrained, in order to allow the server to do the best job it can in performing the search.

The DAV:contains operator evaluates to a Boolean value. It evaluates to TRUE if the content of the resource satisfies the search. Otherwise, It evaluates to FALSE.

Within the DAV:contains XML element, the client provides a phrase: a single word or whitespace delimited sequence of words. Servers MAY ignore punctuation in a phrase. Case-sensitivity is left to the server.

The following things may or may not be done as part of the search: Phonetic methods such as "soundex" may or may not be used. Word stemming may or may not be performed. Thesaurus expansion of words may or may not be done. Right or left truncation may or may not be performed. The search may be case insensitive or case sensitive. The word or words may or may not be interpreted as names. Multiple words may or may not be required to be adjacent or "near" each other. Multiple words may or may not be required to occur in the same order. Multiple words may or may not be treated as a phrase. The search may or may not be interpreted as a request to find documents "similar" to the string operand.

The DAV:score property is intended to be useful to rank documents satisfying the DAV:contains operator.

5.13.1 Examples

The example below shows a search for the phrase "Peter Forsberg".

Depending on its support for content-based searching, a server MAY treat this as a search for documents that contain the words "Peter" and "Forsberg".

```
<D:where>
  <D:contains>Peter Forsberg</D:contains>
</D:where>
```

The example below shows a search for resources that contain "Peter" and "Forsberg".

```
<D:where>
  <D:and>
    <D:contains>Peter</D:contains>
    <D:contains>Forsberg</D:contains>
  </D:and>
</D:where>
```

5.14 The **DAV:limit** XML Element

```
<!ELEMENT limit (nresults) >
```

The **DAV:limit** XML element contains requested limits from the client to limit the size of the reply or amount of effort expended by the server.

5.15 The **DAV:nresults** XML Element

```
<!ELEMENT nresults (#PCDATA)> ;only digits
```

The **DAV:nresults** XML element contains a requested maximum number of records to be returned in a reply. The server MAY disregard this limit. The value of this element is an integer.

5.16 The **DAV:casesensitive** XML attribute

The **DAV:casesensitive** attribute allows clients to specify case-sensitive or case-insensitive behavior for **DAV:basicsearch** operators.

The possible values for **DAV:casesensitive** are "1" or "0". The "1" value indicates case-sensitivity. The "0" value indicates case-insensitivity. The default value is server-specified.

Support for the **DAV:casesensitive** is optional. A server should respond with an error 422 if the **DAV:casesensitive** attribute is used but cannot be supported.

5.17 The **DAV:score** Property

```
<!ELEMENT score (#PCDATA)>
```

The **DAV:score** XML element is a synthetic property whose value is defined only in the context of a query result where the server computes a score, e.g. based on relevance. It may be used in **DAV:select** or **DAV:orderby** elements. Servers SHOULD support this property. The value is a string representing the score, an integer from zero to 10000 inclusive, where a higher value indicates a higher score (e.g. more relevant).

Clients should note that, in general, it is not meaningful to compare the numeric values of scores from two different queries unless both were executed by the same underlying search system on the same collection of resources.

5.18 The `DAV:iscollection` Property

```
<!ELEMENT iscollection (#PCDATA)>
```

The `DAV:iscollection` XML element is a synthetic property whose value is defined only in the context of a query.

The property is TRUE (the literal string "1") of a resource if and only if a `PROPFIND` of the `DAV:resourcetype` property for that resource would contain the `DAV:collection` XML element. The property is FALSE (the literal string "0") otherwise.

Rationale: This property is provided in lieu of defining generic structure queries, which would suffice for this and for many more powerful queries, but seems inappropriate to standardize at this time.

5.18.1 Example of `DAV:iscollection`

This example shows a search criterion that picks out all and only the resources in the scope that are collections.

```
<D:where>
  <D:eq>
    <D:prop><D:iscollection></D:prop>
    <D:literal>1<D:literal>
  </D:eq>
</D:where>
```

5.19 QuerySchema for `DAV:basicsearch`

The `DAV:basicsearch` grammar defines a search criteria that is a Boolean-valued expression, and allows for an arbitrary set of properties to be included in the result record. The result set may be sorted on a set of property values. Accordingly the DTD for schema discovery for this grammar allows the server to express:

1. the set of optional operators defined by the resource.

5.19.1 DTD for `DAV:basicsearch` QSD

```
<!ELEMENT basicsearchschema (properties, operators)>
<!ELEMENT properties (propdesc*)>
<!ELEMENT propdesc (prop, ANY)>
<!ELEMENT operators (opdesc*)>
<!ELEMENT opdesc ANY>
<!ELEMENT operand_property EMPTY>
<!ELEMENT operand_literal EMPTY>
```

The `DAV:properties` element holds a list of descriptions of properties.

The `DAV:operators` element describes the optional operators that may be used in a `DAV:where` element.

5.19.2 `DAV:propdesc` Element

Each instance of a `DAV:propdesc` element describes the property or properties in the `DAV:prop` element it contains. All subsequent elements are descriptions that apply to those properties. All descriptions are optional and may appear in any order. Servers **SHOULD** support all the descriptions defined here, and **MAY** define others.

DASL defines five descriptions. The first, `DAV:datatype`, provides a hint about the type of the property value, and may be useful to a user interface prompting for a value. The remaining four (`DAV:searchable`, `DAV:selectable`, `DAV:sortable`, and `DAV:casesensitive`) identify portions of the query (`DAV:where`, `DAV:select`, and `DAV:orderby`, respectively). If a property has a description for a section, then the server **MUST** allow the property to be used in that section. These descriptions are optional. If a property does not have such a description, or is not described at all, then the server **MAY** still allow the property to be used in the corresponding section.

5.19.3 The `DAV:datatype` Property Description

The `DAV:datatype` element contains a single XML element that provides a hint about the domain of the property, which may be useful to a user interface prompting for a value to be used in a query. The namespace for expressing a DASL defined data type is "urn:uuid:C2F41010-65B3-11d1-A29F-00AA00C14882/".

<!--ELEMENT datatype

ANY -->

DASL defines the following data type elements:

Name	example
boolean	1, 0
string	FooBar
dateTime.iso8601tz	1994-11-05T08:15:5Z
float	.314159265358979E+1
int	-259, 23

If the data type of a property is not given, then the data type defaults to string.

5.19.4 The `DAV:searchable` Property Description

<!--ELEMENT searchable

EMPTY -->

If this element is present, then the server **MUST** allow this property to appear within a `DAV:where` element where an operator allows a property. Allowing a search does not mean that the property is guaranteed to be defined on every resource in the scope, it only indicates the server's willingness to check.

5.19.5 The `DAV:selectable` Property Description

<!--ELEMENT selectable

EMPTY -->

This element indicates that the property may appear in the `DAV:select` element.

5.19.6 The DAV:sortable Property Description

This element indicates that the property may appear in the DAV:orderby element

```
<!ELEMENT sortable EMPTY >
```

5.19.7 The DAV:casesensitive Property Description

This element only applies to properties whose data type is "string" as per the DAV:datatype property description. Its presence indicates that compares performed for searches, and the comparisons for ordering results on the string property will be case sensitive. (The default is case insensitive.)

```
<!ELEMENT casesensitive EMPTY >
```

5.19.8 The DAV:operators XML Element

The DAV:operators element describes every optional operator supported in a query. (Mandatory operators are not listed since they are mandatory and permit no variation in syntax.). All optional operators that are supported MUST be listed in the DAV:operators element. The listing for an operator consists of the operator (as an empty element), followed by one element for each operand. The operand MUST be either DAV:operand_property or DAV:operand_literal, which indicate that the operand in the corresponding position is a property or a literal value, respectively. If an operator is polymorphic (allows more than one operand syntax) then each permitted syntax MUST be listed separately.

```
<D:propdesc><D:like/><D:operand_property/><D:operand_literal/></D:propdesc>
```

5.19.9 Example of Query Schema for DAV:basicsearch

```
<D:basicsearchschema xmlns:D="DAV:" xmlns:t="urn:uuid:C2F41010-65B3-11d1-A29F-C
  <D:properties>
    <D:propdesc>
      <D:prop><D:getcontentlength/></D:prop>
      <D:datatype><t:int></D:datatype>
      <D:searchable/><D:selectable/><D:sortable/>
    </D:propdesc>
    <D:propdesc>
      <D:prop><D:getcontenttype/><D:displayname></D:prop>
      <D:searchable/><D:selectable/> <D:sortable/>
    </D:propdesc>
    <D:propdesc>
      <D:prop><J:fstop/></D:prop>
      <D:selectable/>
    </D:propdesc>
  </D:properties>
  <D:operators>
    <D:opdesc>
      <D:isdefined/><D:operand_property/>
    </D:opdesc>
    <D:opdesc>
      <D:like/><D:operand_property/><D:operand_literal/>
    </D:opdesc>
  </D:operators>
</D:basicsearchschema>
```

This response lists four properties. The datatype of the last three properties is not given, so it

defaults to string. All are selectable, and the first three may be searched. All but the last may be used in a sort. Of the optional DAV operators, `DAV:isdefined` and `DAV:like` are supported.

Note: The schema discovery defined here does not provide for discovery of supported values of the `DAV:casesensitive` attribute. This may require that the reply also list the mandatory operators.

6 Internationalization Considerations

Clients have the opportunity to tag properties when they are stored in a language. The server **SHOULD** read this language-tagging by examining the `xml:lang` attribute on any properties stored on a resource.

The `xml:lang` attribute specifies a nationalized collation sequence when properties are compared.

Comparisons when this attribute differs have undefined order.

7 Security Considerations

This section is provided to detail issues concerning security implications of which DASL applications need to be aware. All of the security considerations of HTTP/1.1 also apply to DASL. In addition, this section will include security risks inherent in searching and retrieval of resource properties and content.

A query must not allow one to retrieve information about values or existence of properties that one could not obtain via `PROPFIND`. (e.g. by use in `DAV:orderby`, or in expressions on properties.)

A server should prepare for denial of service attacks. For example a client may issue a query for which the result set is expensive to calculate or transmit because many resources match or must be evaluated.

7.1 Implications of XML External Entities

XML supports a facility known as "external entities", defined in section 4.2.2 of [REC-XML], which instruct an XML processor to retrieve and perform an inline include of XML located at a particular URI. An external XML entity can be used to append or modify the document type declaration (DTD) associated with an XML document. An external XML entity can also be used to include XML within the content of an XML document. For non-validating XML, such as the XML used in this specification, including an external XML entity is not required by [REC-XML]. However, [REC-XML] does state that an XML processor may, at its discretion, include the external XML entity.

External XML entities have no inherent trustworthiness and are subject to all the attacks that are endemic to any HTTP GET request. Furthermore, it is possible for an external XML entity to modify the DTD, and hence affect the final form of an XML document, in the worst case significantly modifying its semantics, or exposing the XML processor to the security risks discussed in [RFC2376]. Therefore, implementers must be aware that external XML entities should be treated as untrustworthy.

There is also the scalability risk that would accompany a widely deployed application which made use of external XML entities. In this situation, it is possible that there would be significant numbers of requests for one external XML entity, potentially overloading any server which fields

requests for the resource containing the external XML entity.

8 Scalability

Query grammars are identified by URIs. Applications SHOULD not attempt to retrieve these URIs even if they appear to be retrievable (for example, those that begin with "http://")

9 Authentication

Authentication mechanisms defined in WebDAV will also apply to DASL.

10 IANA Considerations

This document uses the namespace defined by [WebDAV] for XML elements. All other IANA considerations mentioned in [WebDAV] also applicable to DASL

11 Copyright

To be supplied.

12 Intellectual Property

To be supplied.

13 References

13.1 Normative References

[DASLREQ] J. Davis, S. Reddy, J. Slein, "Requirements for DAV Searching and Locating", Feb 24, 1999, internet-draft, work-in-progress, draft-dasl-requirements-01.txt

[RFC2068] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2068, U.C. Irvine, DEC, MIT/LCS, January 1997.

[RFC2119] S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels." RFC 2119, BCP 14. Harvard University. March, 1997.

[RFC2376] E. Whitehead, M. Murata, "XML Media Types". RFC 2376, July 1998.

[WebDAV] Y. Goland, E.J. Whitehead, A. Faizi, S.R. Carter, D. Jenson, "HTTP Extensions for Distributed Authoring -- WebDAV", RFC 2518, February 1999.

[XML] T. Bray, J. Paoli, C. M. Sperberg-McQueen, "Extensible Markup Language (XML) 1.0", September 16, 1998, W3C Recommendation.

[XMLNS] T. Bray, D. Hollander, A. Layman, "Namespaces in XML", 14-January-1999, W3C

Recommendation. <http://www.w3.org/TR/REC-xml-names/>.

13.2 Non-Normative References

[ANSISQL] ANSI, "Information Systems - Database Language - SQL (includes ANSI X3.168-1989)", ANSI X3.135-1992 (R1998), 1992.

14 Author's Addresses

Saveen Reddy
Microsoft
One Microsoft Way
Redmond WA, 9085-6933
Email: saveenr@microsoft.com

Dale Lowry
Novell
1555 N. Technology Way
M/S ORM-M-314
Orem, UT 84097
Email: dlowry@novell.com

Surendra Reddy
Oracle Corporation
600 Oracle Parkway, M/S 6op3,
Redwoodshores, CA 94065
Email: skreddy@us.oracle.com
Phone: (650) 506 5441

Rick Henderson
Netscape
Email: rickh@netscape.com

Jim Davis
CourseNet Systems
San Francisco, CA
Email: jrd3@alum.mit.edu

Alan Babich
Filenet
3565 Harbor Blvd.
Costa Mesa, CA 92626
714-966-3403
Email: ababich@filenet.com

15 APPENDICES

Three-Valued Logic in `DAV:basicsearch`

ANSI standard three valued logic is used when evaluating the search condition (as defined in the ANSI standard SQL specifications, for example in ANSI X3.135-1992, section 8.12, pp. 188-189, section 8.2, p. 169, General Rule 1)a), etc.).

ANSI standard three valued logic is undoubtedly the most widely practiced method of dealing with the issues of properties in the search condition not having a value (e.g., being null or not defined) for the resource under scan, and with undefined expressions in the search condition (e.g., division by zero, etc.). Three valued logic works as follows.

Undefined expressions are expressions for which the value of the expression is not defined. Undefined expressions are a completely separate concept from the truth value UNKNOWN, which is, in fact, well defined. Property names and literal constants are considered expressions for purposes of this section. If a property in the current resource under scan has not been set to a value (either because the property is not defined for the current resource, or because it is null for the current resource), then the value of that property is undefined for the resource under scan. DASL 1.0 has no arithmetic division operator, but if it did, division by zero would be an undefined arithmetic expression.

If any subpart of an arithmetic, string, or datetime subexpression is undefined, the whole arithmetic, string, or datetime subexpression is undefined.

There are no manifest constants to explicitly represent undefined number, string, or datetime values.

Since a Boolean value is ultimately returned by the search condition, arithmetic, string, and datetime expressions are always arguments to other operators. Examples of operators that convert arithmetic, string, and datetime expressions to Boolean values are the six relational operators ("greater than", "less than", "equals", etc.). If either or both operands of a relational operator have undefined values, then the relational operator evaluates to UNKNOWN. Otherwise, the relational operator evaluates to TRUE or FALSE, depending upon the outcome of the comparison.

The Boolean operators DAV:and, DAV:or and DAV:not are evaluated according to the following rules:

UNKNOWN and UNKNOWN = UNKNOWN

UNKNOWN or UNKNOWN = UNKNOWN

not UNKNOWN = UNKNOWN

UNKNOWN and TRUE = UNKNOWN

UNKNOWN and FALSE = FALSE

UNKNOWN and UNKNOWN = UNKNOWN

UNKNOWN or TRUE = TRUE

UNKNOWN or FALSE = UNKNOWN

UNKNOWN or UNKNOWN = UNKNOWN

16 Change History

Feb 14, 1998

Initial Draft

Feb 28, 1998

Referring to DASL as an extension to HTTP/1.1 rather than DAV

Added new sections "Notational Conventions", "Protocol Model", "Security Considerations"

Changed section 3 to "Elements of Protocol"

Added some stuff to introduction

Added "result set" terminology

Added "IANA Considerations".

Mar 9, 1998

Moved sub-headings of "Elements of Protocol" to first level and removed "Elements of Protocol" Heading.

Added an sentence in introduction explaining that this is a "sketch" of a protocol.

Mar 11, 1998

Added orderby, data typing, three valued logic, query schema property, and element definitions for schema for basicsearch.

April 8, 1998

- made changes based on last week's DASL BOF.

May 8, 1998

Removed most of DAV:searcherror; converted to DAV:searchredirect

Altered DAV:basicsearch grammar to use avoid use of ANY in DTD

June 17, 1998

-Added details on Query Schema Discovery

-Shortened list of data types

June 23, 1998

moved data types before change history

rewrote the data types section

removed the casesensitive element and replace with the casesensitive attribute

added the casesensitive attribute to the DTD for all operations that might work on a string

Jul 20, 1998

A series of changes. See Author's meeting minutes for details.

July 28, 1998

Changes as per author's meeting. QSD uses SEARCH, not PROPFIND.

Moved text around to keep concepts nearby.

Boolean literals are 1 and 0, not T and F.

contains changed to contentspassthrough.

Renamed rank to score.

July 28, 1998

Added Dale Lowry as Author

September 4, 1998

Added 422 as response when query lists unimplemented operators.

DAV:literal declares a default value for xml:space, 'preserve' (see XML spec, section 2.10)

moved to new XML namespace syntax

September 22, 1998

Changed "simplesearch" to "basicsearch"

Changed isnull to isdefined

Defined NULLness as having a 404 or 403 response

used ENTITY syntax in DTD

Added redirect

October 9, 1998

Fixed a series of typographical and formatting errors.

Modified the section of three-valued logic to use a table rather than a text description of the role of UNKNOWN in expressions.

November 2, 1998

Added the DAV:contains operator.

Removed the DAV:contentpassthrough operator.

November 18, 1998

Various author comments for submission

June 3, 1999

Cosmetic and minor editorial changes only. Fix nits reported by Jim Whitehead in email of April 26, 1999. Converted to HTML from Word 97, manually.

JAPAN ELECTRONIC INDUSTRY
DEVELOPMENT ASSOCIATION STANDARD

Digital Still Camera Image File Format Standard
(Exchangeable image file format for Digital Still Camera: Exif)

Version 2.1

JEIDA-49-1998

Revised June 1998

Revised October 1997

Established November 1995

JAPAN ELECTRONIC INDUSTRY
DEVELOPMENT ASSOCIATION

Contents

Revision History

Contents

General	1
1.1. Objectives	1
1.2. Scope and Abbreviation	1
1.3. Format Structure	1
1.4. Exif Image File Specification	2
1.5. Exif Audio File Specification	3
1.6. Relation between Image and Audio File Specification	4
1.7. Presupposed Systems and Compatibility	5
Exif Image File Specification	6
2.1. Outline of the Exif Image File Specification	6
2.2. Format Version	6
2.3. Definition of Glossary	6
2.4. Specifications Relating to Image Data	8
2.4.1. Number of Pixels	8
2.4.2. Pixel Aspect	8
2.4.3. Pixel Composition and Sampling	8
2.4.4. Image Data Arrangement	10
2.5. Basic Structure of Image Data	11
2.5.1. Basic Structure of Primary Image Data	11
2.5.2. Basic Structure of Uncompressed RGB Data	11
2.5.3. Basic Structure of YCbCr Uncompressed Data	13
2.5.4. Basic Structure of JPEG Compressed Data	14
2.5.5. Basic Structure of Thumbnail Data	15
2.6. Tags	17
2.6.1. Features of Attribute Information	17
2.6.2. IFD Structure	17
2.6.3. Exif-specific IFD	19
2.6.4. TIFF Rev. 6.0 Attribute Information	21
2.6.5. Exif IFD Attribute Information	34
2.6.6. GPS Attribute Information	56
2.6.7. Interoperability IFD Attribute Information	65
2.6.8. Tag Support Levels	66
2.7. JPEG Marker Segments Used in Exif	70
2.7.1. JPEG Marker Segments	70
2.7.2. Interoperability Structure of APP1 in Compressed Data	76

2.7.3.	Interoperability Structure of APP2 in Compressed Data	78
2.8.	Data Description	83
2.8.1.	Stipulations on Compressed Image Size	83
2.8.2.	Stipulations on Thumbnails.....	87
2.8.3.	File Name Stipulations	87
2.8.4.	Byte Order Stipulations	87
3.	Exif Audio File Specification	88
3.1.	Outline of the Exif Audio File Specification	88
3.2.	Format Version.....	88
3.3.	Definition of Terms	88
3.4.	Specifications Relating to Audio Data	89
3.4.1.	Sampling Frequency	89
3.4.2.	Bit Size	89
3.4.3.	Channels.....	89
3.4.4.	Compression Schemes	89
3.5.	Basic Structure of Audio Data.....	90
3.5.1.	Basic Structure of WAVE Form Audio Files.....	90
3.5.2.	Basic Structure of PCM Audio Data.....	100
3.5.3.	Basic Structure of μ -Law Audio Data.....	103
3.5.4.	Basic Structure of IMA-ADPCM Audio Data.....	105
3.6.	Chunks Used	109
3.6.1.	WAVE Form Audio File Basic Chunks	109
3.6.2.	LIST Chunk and INFO List.....	110
3.6.3.	Chunks for Attribute Information Specific to Exif Audio Files	116
3.7.	Data Description	122
3.7.1.	File Naming Stipulation.....	122
付録 A	Image File Description Examples	125
A.1	Uncompressed RGB File.....	125
A.2	Uncompressed YCbCr File	128
A.3	JPEG Compressed (4:2:2) File	133
A.4	JPEG Compressed (4:2:0) File	138
付録 B	Audio File Description Examples	144
B.1	PCM Audio Data	144
	-Law Audio Data	146
B.3	IMA-ADPCM Audio Data	149
付録 C	APEX Units	152
付録 D	Recommended Implementation Examples	153
D.1	Recommended Directory Name Usage Examples.....	153
D.2	Recommended File Naming Usage Examples.....	153

D.3	Recommended File Operation Usage Examples.....	153
D.4	Interoperability "Recommended Exif Interoperability Rules" (ExifR98)	154
付録 E	Color Space Guidelines.....	155
E.1	sRGB	155
E.2	Tone Reproduction (Brightness and Contrast)	156
E.3	Luminance/Chrominance and RGB Transformation	157
付録 F	Notes on Conversion to FlashPix.....	158
F.1	Converting Image Data.....	160
F.2	Converting Tag Data	163
F.3	Converting to FlashPix Extensions (APP2).....	166
References	166

2.6. Tags

2.6.1. Features of Attribute Information

RGB data conforms to Baseline TIFF Rev. 6.0 RGB Full Color Images, and YCbCr data to TIFF Rev. 6.0 Extensions YCbCr Images. Accordingly, the parts that follow the TIFF structure must be recorded in conformance to the TIFF standard. In addition to the attribute information indicated as mandatory in the TIFF standard, this Exif standard adds the TIFF optional tags that can be used in a DSC or other system, Exif-specific tags for recording DSC-specific attribute information, and GPS tags for recording position information. There are also Exif-original specifications not found in the TIFF standard for compressed recording of thumbnails.

Recording of compressed data differs from uncompressed data in the following respects:

- When the primary image data is recorded in compressed form, there is no tag indicating the primary image itself or its address (pointer),
- When thumbnail data is recorded in compressed form, address and size are designated using Exif-specific tags,
- Tags that duplicate information given in the JPEG Baseline are not recorded (for either primary images or thumbnails).
- Information relating to compression can be recorded using the tags for this purpose.

2.6.2. IFD Structure

The IFD used in this standard consists of a 2-byte count (number of fields), 12-byte field Interoperability arrays, and 4-byte offset to the next IFD, in conformance with TIFF Rev. 6.0.

Each of the 12-byte field Interoperability consists of the following four elements respectively.

Bytes 0-1	Tag
Bytes 2-3	Type
Bytes 4-7	Count
Bytes 8-11	Value Offset

Each element is explained briefly below. For details see TIFF Rev. 6.0.

Tag

Each tag is assigned a unique 2-byte number to identify the field. The tag numbers in the Exif 0th IFD and 1st IFD are all the same as the TIFF tag numbers.

Type

The following types are used in Exif:

- | | |
|----------------|--|
| 1 = BYTE | An 8-bit unsigned integer., |
| 2 = ASCII | An 8-bit byte containing one 7-bit ASCII code. The final byte is terminated with NULL., |
| 3 = SHORT | A 16-bit (2-byte) unsigned integer, |
| 4 = LONG | A 32-bit (4-byte) unsigned integer, |
| 5 = RATIONAL | Two LONGs. The first LONG is the numerator and the second LONG expresses the denominator., |
| 7 = UNDEFINED | An 8-bit byte that can take any value depending on the field definition, |
| 9 = SLONG | A 32-bit (4-byte) signed integer (2's complement notation), |
| 10 = SRATIONAL | Two SLONGs. The first SLONG is the numerator and the second SLONG is the denominator. |

Count

The number of values. It should be noted carefully that the count is not the sum of the bytes. In the case of one value of SHORT (16 bits), for example, the count is '1' even though it is 2 bytes.

Value Offset

This tag records the offset from the start of the TIFF header to the position where the value itself is recorded. In cases where the value fits in 4 bytes, the value itself is recorded. If the value is smaller than 4 bytes, the value is stored in the 4-byte area starting from the left, i.e., from the lower end of the byte offset area. For example, in big endian format, if the type is SHORT and the value is 1, it is recorded as 00010000.H.

Note that field Interoperability must be recorded in sequence starting from the smallest tag number. There is no stipulation regarding the order or position of tag value (Value) recording.

2.6.3. Exif-specific IFD

A. Exif IFD

Exif IFD is a set of tags for recording Exif-specific attribute information. It is pointed to by the offset from the TIFF header (Value Offset) indicated by an Exif private tag value.

Exif IFD Pointer

Tag	= 34665 (8769.H)
Type	= LONG
Count	= 1
Default	= none

A pointer to the Exif IFD. Interoperability, Exif IFD has the same structure as that of the IFD specified in TIFF. Ordinarily, however, it does not contain image data as in the case of TIFF.

B. GPS IFD

GPS IFD is a set of tags for recording GPS information. It is pointed to by the offset from the TIFF header (Value Offset) indicated by a GPS private tag value.

GPS Info IFD Pointer

Tag	= 34853 (8825.H)
Type	= LONG
Count	= 1
Default	= none

A pointer to the GPS Info IFD. The Interoperability structure of the GPS Info IFD, like that of Exif IFD, has no image data.

C. Interoperability IFD

Interoperability IFD is composed of tags which stores the information to ensure the Interoperability and pointed by the following tag located in Exif IFD.

Interoperability IFD Pointer

Tag	= 40965 (A005.H)
Type	= LONG
Count	= 1
Default	= None

The Interoperability structure of Interoperability IFD is same as TIFF defined IFD structure but does

not contain the image data characteristically compared with normal TIFF IFD.

2.6.4. TIFF Rev. 6.0 Attribute Information

Table 3 lists the attribute information used in Exif, including the attributes given as mandatory in Baseline TIFF Rev. 6.0 RGB Full Color Images and TIFF Rev. 6.0 Extensions YCbCr Images, as well as the optional TIFF tags used by DSC and other systems. The contents are explained below.

Table 3 TIFF Rev. 6.0 Attribute Information Used in Exif

Tag Name	Field Name	Tag ID		Type	Count
		Dec	Hex		
A. Tags relating to image data structure					
Image width	ImageWidth	256	100	SHORT or LONG	1
Image height	ImageLength	257	101	SHORT or LONG	1
Number of bits per component	BitsPerSample	258	102	SHORT	3
Compression scheme	Compression	259	103	SHORT	1
Pixel composition	PhotometricInterpretation	262	106	SHORT	1
Orientation of image	Orientation	274	112	SHORT	1
Number of components	SamplesPerPixel	277	115	SHORT	1
Image data arrangement	PlanarConfiguration	284	11C	SHORT	1
Subsampling ratio of Y to C	YCbCrSubSampling	530	212	SHORT	2
Y and C positioning	YCbCrPositioning	531	213	SHORT	1
Image resolution in width direction	XResolution	282	11A	RATIONAL	1
Image resolution in height direction	YResolution	283	11B	RATIONAL	1
Unit of X and Y resolution	ResolutionUnit	296	128	SHORT	1
B. Tags relating to recording offset					
Image data location	StripOffsets	273	111	SHORT or LONG	*S
Number of rows per strip	RowsPerStrip	278	116	SHORT or LONG	1
Bytes per compressed strip	StripByteCounts	279	117	SHORT or LONG	*S
Offset to JPEG SOI	JPEGInterchangeFormat	513	201	LONG	1
Bytes of JPEG data	JPEGInterchangeFormatLength	514	202	LONG	1
C. Tags relating to image data characteristics					
Transfer function	TransferFunction	301	12D	SHORT	3 * 256
White point chromaticity	WhitePoint	318	13E	RATIONAL	2
Chromaticities of primaries	PrimaryChromaticities	319	13F	RATIONAL	6
Color space transformation matrix coefficients	YCbCrCoefficients	529	211	RATIONAL	3
Pair of black and white reference values	ReferenceBlackWhite	532	214	RATIONAL	6
D. Other tags					
File change date and time	DateTime	306	132	ASCII	20
Image title	ImageDescription	270	10E	ASCII	Any
Image input equipment manufacturer	Make	271	10F	ASCII	Any
Image input equipment model	Model	272	110	ASCII	Any
Software used	Software	305	131	ASCII	Any
Person who created the image	Artist	315	13B	ASCII	Any
Copyright holder	Copyright	3432	8298	ASCII	Any

*S Chunky format: StripsPerImage

Planar format: SamplesPerImage * StripsPerImage

StripsPerImage = floor((ImageLength + RowsPerStrip - 1) / RowsPerStrip)

A. Tags relating to image data structure

ImageWidth

The number of columns of image data, equal to the number of pixels per row. In JPEG compressed data a JPEG marker is used instead of this tag.

Tag = 256 (100.H)
Type = SHORT or LONG
Count = 1
Default = none

ImageLength

The number of rows of image data. In JPEG compressed data a JPEG marker is used instead of this tag.

Tag = 257 (101.H)
Type = SHORT or LONG
Count = 1
Default = none

BitsPerSample

The number of bits per image component. In this standard each component of the image is 8 bits, so the value for this tag is 8. See also *SamplesPerPixel*. In JPEG compressed data a JPEG marker is used instead of this tag.

Tag = 258 (102.H)
Type = SHORT
Count = 3
Default = 8 8 8

Compression

The compression scheme used for the image data. When a primary image is JPEG compressed, this designation is not necessary and is omitted. When thumbnails use JPEG compression, this tag value is set to 6.

Tag = 259 (103.H)
Type = SHORT
Count = 1
Default = none
1 = uncompressed
6 = JPEG compression (thumbnails only)

Other = reserved

PhotometricInterpretation

The pixel composition. In JPEG compressed data a JPEG marker is used instead of this tag.

Tag = 262 (106.H)

Type = SHORT

Count = 1

Default = none

2 = RGB

6 = YCbCr

Other = reserved

Orientation

The image orientation viewed in terms of rows and columns.

Tag = 274 (112.H)

Type = SHORT

Count = 1

Default = 1

1 = The 0th row is at the visual top of the image, and the 0th column is the visual left-hand side.

2 = The 0th row is at the visual top of the image, and the 0th column is the visual right-hand side.

3 = The 0th row is at the visual bottom of the image, and the 0th column is the visual right-hand side.

4 = The 0th row is at the visual bottom of the image, and the 0th column is the visual left-hand side.

5 = The 0th row is the visual left-hand side of of the image, and the 0th column is the visual top.

6 = The 0th row is the visual right-hand side of of the image, and the 0th column is the visual top.

7 = The 0th row is the visual right-hand side of of the image, and the 0th column is the visual bottom.

8 = The 0th row is the visual left-hand side of of the image, and the 0th column is the visual bottom.

Other = reserved

SamplesPerPixel

The number of components per pixel. Since this standard applies to RGB and YCbCr images, the value set for this tag is 3. In JPEG compressed data a JPEG marker is used instead of this tag.

Tag = 277 (115.H)

Type = SHORT

Count = 1

Default = 3

PlanarConfiguration

Indicates whether pixel components are recorded in chunky or planar format. In JPEG compressed files a JPEG marker is used instead of this tag. If this field does not exist, the TIFF default of 1 (chunky) is assumed.

Tag = 284 (11C.H)
Type = SHORT
Count = 1
1 = chunky format
2 = planar format
Other = reserved

YCbCrSubSampling

The sampling ratio of chrominance components in relation to the luminance component. In JPEG compressed data a JPEG marker is used instead of this tag.

Tag = 530 (212.H)
Type = SHORT
Count = 2
[2, 1] = YCbCr4:2:2
[2, 2] = YCbCr4:2:0
Other = reserved

YCbCrPositioning

The position of chrominance components in relation to the luminance component. This field is designated only for JPEG compressed data or uncompressed YCbCr data. The TIFF default is 1 (centered); but when Y:Cb:Cr = 4:2:2 it is recommended in this standard that 2 (co-sited) be used to record data, in order to improve the image quality when viewed on TV systems. When this field does not exist, the reader shall assume the TIFF default. In the case of Y:Cb:Cr = 4:2:0, the TIFF default (centered) is recommended. If the reader does not have the capability of supporting both kinds of *YCbCrPositioning*, it shall follow the TIFF default regardless of the value in this field. It is preferable that readers be able to support both centered and co-sited positioning.

Tag = 531 (213.H)
Type = SHORT
Count = 1
Default = 1
1 = centered

2 = co-sited
Other = reserved

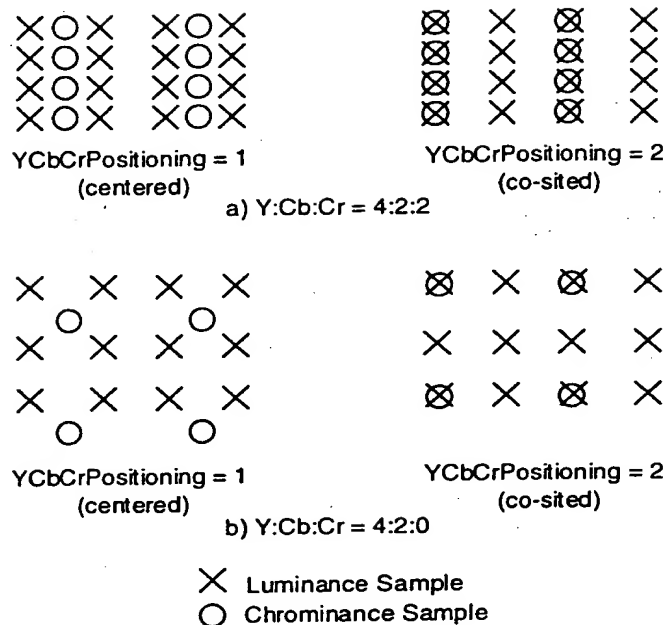


Fig. 8 YCbCrPositioning

XResolution

The number of pixels per *ResolutionUnit* in the *ImageWidth* direction. When the image resolution is unknown, 72 [dpi] is designated.

Tag = 282 (11A.H)
Type = RATIONAL
Count = 1
Default = 72

YResolution

The number of pixels per *ResolutionUnit* in the *ImageLength* direction. The same value as *XResolution* is designated.

Tag = 283 (11B.H)
Type = RATIONAL
Count = 1
Default = 72

ResolutionUnit

The unit for measuring *XResolution* and *YResolution*. The same unit is used for both *XResolution* and

YResolution. If the image resolution is unknown, 2 (inches) is designated.

Tag = 296 (128.H)

Type = SHORT

Count = 1

Default = 2

2 = inches

3 = centimeters

Other = reserved

B. Tags relating to recording offset

StripOffsets

For each strip, the byte offset of that strip. It is recommended that this be selected so the number of strip bytes does not exceed 64 Kbytes. With JPEG compressed data this designation is not needed and is omitted. See also *RowsPerStrip* and *StripByteCounts*.

Tag	=	273 (111.H)	
Type	=	SHORT or LONG	
Count	=	StripsPerImage	(when PlanarConfiguration = 1)
	=	SamplesPerPixel * StripsPerImage	(when PlanarConfiguration = 2)
Default	=	none	

RowsPerStrip

The number of rows per strip. This is the number of rows in the image of one strip when an image is divided into strips. With JPEG compressed data this designation is not needed and is omitted. See also *RowsPerStrip* and *StripByteCounts*.

Tag

Tag	=	278 (116.H)
Type	=	SHORT or LONG
Count	=	1
Default	=	none

StripByteCounts

The total number of bytes in each strip. With JPEG compressed data this designation is not needed and is omitted.

Tag	=	279 (117.H)	
Type	=	SHORT or LONG	
Count	=	StripsPerImage	(when PlanarConfiguration = 1)
	=	SamplesPerPixel * StripsPerImage	(when PlanarConfiguration = 2)
Default	=	none	

JPEGInterchangeFormat

The offset to the start byte (SOI) of JPEG compressed thumbnail data. This is not used for primary image JPEG data.

Tag	=	513 (201.H)
Type	=	LONG

Default = none

JPEGInterchangeFormatLength

The number of bytes of JPEG compressed thumbnail data. This is not used for primary image JPEG data. JPEG thumbnails are not divided but are recorded as a continuous JPEG bitstream from SOI to EOI. APPn and COM markers should not be recorded. Compressed thumbnails must be recorded in no more than 64 Kbytes, including all other data to be recorded in APP1.

Tag = 514 (202.H)

Type = LONG

Default = none

C. Tags Relating to Image Data Characteristics

TransferFunction

A transfer function for the image, described in tabular style. Normally this tag is not necessary, since color space is specified in the color space information tag (*ColorSpace*).

Tag = 301 (12D.H)
Type = SHORT
Count = 3 * 256
Default = none

WhitePoint

The chromaticity of the white point of the image. Normally this tag is not necessary, since color space is specified in the color space information tag (*ColorSpace*).

Tag = 318 (13E.H)
Type = RATIONAL
Count = 2
Default = none

PrimaryChromaticities

The chromaticity of the three primary colors of the image. Normally this tag is not necessary, since color space is specified in the color space information tag (*ColorSpace*).

Tag = 319 (13F.H)
Type = RATIONAL
Count = 6
Default = none

YCbCrCoefficients

The matrix coefficients for transformation from RGB to YCbCr image data. No default is given in TIFF; but here the value given in Appendix E, "Color Space Guidelines," is used as the default. The color space is declared in a color space information tag, with the default being the value that gives the optimal image characteristics Interoperability this condition.

Tag = 529 (211.H)
Type = RATIONAL
Count = 3
Default = See Appendix E.

ReferenceBlackWhite

The reference black point value and reference white point value. No defaults are given in TIFF, but the values below are given as defaults here. The color space is declared in a color space information tag, with the default being the value that gives the optimal image characteristics Interoperability these conditions.

Tag = 532 (214.H)

Type = RATIONAL

Count = 6

Default = [0, 255, 0, 255, 0, 255] (when PhotometricInterpretation is RGB)

[0, 255, 0, 128, 0, 128] (when PhotometricInterpretation is YCbCr)

D. Other Tags

DateTime

The date and time of image creation. In this standard it is the date and time the file was changed. The format is "YYYY:MM:DD HH:MM:SS" with time shown in 24-hour format, and the date and time separated by one blank character [20.H]. When the date and time are unknown, all the character spaces except colons (":") may be filled with blank characters, or else the Interoperability field may be filled with blank characters. The character string length is 20 bytes including NULL for termination. When the field is left blank, it is treated as unknown.

Tag	=	306 (132.H)
Type	=	ASCII
Count	=	20
Default	=	none

ImageDescription

A character string giving the title of the image. It may be a comment such as "1988 company picnic" or the like. Two-byte character codes cannot be used. When a 2-byte code is necessary, the Exif Private tag *UserComment* is to be used.

Tag	=	270 (10E.H)
Type	=	ASCII
Count	=	Any
Default	=	none

Make

The manufacturer of the recording equipment. This is the manufacturer of the DSC, scanner, video digitizer or other equipment that generated the image. When the field is left blank, it is treated as unknown.

Tag	=	271 (10F.H)
Type	=	ASCII
Count	=	Any
Default	=	none

Model

The model name or model number of the equipment. This is the model name of number of the DSC, scanner, video digitizer or other equipment that generated the image. When the field is left blank, it

is treated as unknown.

Tag = 272 (110.H)
Type = ASCII
Count = Any
Default = none

Software

This tag records the name and version of the software or firmware of the camera or image input device used to generate the image. The detailed format is not specified, but it is recommended that the example shown below be followed. When the field is left blank, it is treated as unknown.

Ex.) "Exif Software Version 1.00a"

Tag = 305 (131h)
Type = ASCII
Count = Any
Default = none

Artist

This tag records the name of the camera owner, photographer or image creator. The detailed format is not specified, but it is recommended that the information be written as in the example below for ease of Interoperability. When the field is left blank, it is treated as unknown.

Ex.) "Camera owner, John Smith; Photographer, Michael Brown; Image creator, Ken James"

Tag = 315 (13Bh)
Type = ASCII
Count = Any
Default = none

Copyright

Copyright information. In this standard the tag is used to indicate both the photographer and editor copyrights. It is the copyright notice of the person or organization claiming rights to the image. The Interoperability copyright statement including date and rights should be written in this field; e.g., "Copyright, John Smith, 19xx. All rights reserved." In this standard the field records both the photographer and editor copyrights, with each recorded in a separate part of the statement. When there is a clear distinction between the photographer and editor copyrights, these are to be written in the order of photographer followed by editor copyright, separated by NULL (in this case, since the statement also ends with a NULL, there are two NULL codes) (see example 1). When only the photographer copyright is given, it is terminated by one NULL code (see example 2). When only the

editor copyright is given, the photographer copyright part consists of one space followed by a terminating NULL code, then the editor copyright is given (see example 3). When the field is left blank, it is treated as unknown.

Ex. 1) When both the photographer copyright and editor copyright are given.

Photographer copyright + NULL[00.H] + editor copyright + NULL[00.H]

Ex. 2) When only the photographer copyright is given.

Photographer copyright + NULL[00.H]

Ex. 3) When only the editor copyright is given.

Space[20.H] + NULL[00.H] + editor copyright + NULL[00.H]

Tag = 33432 (8298.H)

Type = ASCII

Count = Any

Default = none

2.6.5. Exif IFD Attribute Information

The attribute information (field names and codes) recorded in the Exif IFD is given in Table 4 and Table 5 followed by an explanation of the contents.

Table 4 Exif IFD Attribute Information (1)

Tag Name	Field Name	Tag ID		Type	Count
		Dec	Hex		
A. Tags Relating to Version					
Exif version	ExifVersion	36864	9000	UNDEFINED	4
Supported FlashPix version	FlashPixVersion	40960	A000	UNDEFINED	4
B. Tag Relating to Image Data Characteristics					
Color space information	ColorSpace	40961	A001	SHORT	1
C. Tags Relating to Image Configuration					
Meaning of each component	ComponentsConfiguration	37121	9101	UNDEFINED	4
Image compression mode	CompressedBitsPerPixel	37122	9102	RATIONAL	1
Valid image width	PixelXDimension	40962	A002	SHORT or LONG	1
Valid image height	PixelYDimension	40963	A003	SHORT or LONG	1
D. Tags Relating to User Information					
Manufacturer notes	MakerNote	37500	927C	UNDEFINED	Any
User comments	UserComment	37510	9286	UNDEFINED	Any
E. Tag Relating to Related File Information					
Related audio file	RelatedSoundFile	40964	A004	ASCII	13
F. Tags Relating to Date and Time					
Date and time of original data generation	DateTimeOriginal	36867	9003	ASCII	20
Date and time of digital data generation	DateTimeDigitized	36868	9004	ASCII	20
Date/Time subseconds	SubSecTime	37520	9290	ASCII	Any
DateTimeOriginal subseconds	SubSecTimeOriginal	37521	9291	ASCII	Any
DateTimeDigitized subseconds	SubSecTimeDigitized	37522	9292	ASCII	Any
G. Tags Relating to Picture-Taking Conditions					
See Table 5					
H. Tags Relating to Date and Time					
Pointer of Interoperability IFD	Interoperability IFD Pointer	40965	A005	LONG	1

Table 5 Exif IFD Attribute Information (2)

G. Tags Relating to Picture-Taking Conditions					
Exposure time	ExposureTime	33434	829A	RATIONAL	1
F number	FNumber	33437	829D	RATIONAL	1
Exposure program	ExposureProgram	34850	8822	SHORT	1
Spectral sensitivity	SpectralSensitivity	34852	8824	ASCII	Any
ISO speed rating	ISOSpeedRatings	34855	8827	SHORT	Any
Optoelectric conversion factor	OECF	34856	8828	UNDEFINED	Any
Shutter speed	ShutterSpeedValue	37377	9201	SRATIONAL	1
Aperture	ApertureValue	37378	9202	RATIONAL	1
Brightness	BrightnessValue	37379	9203	SRATIONAL	1
Exposure bias	ExposureBiasValue	37380	9204	SRATIONAL	1
Maximum lens aperture	MaxApertureValue	37381	9205	RATIONAL	1
Subject distance	SubjectDistance	37382	9206	RATIONAL	1
Metering mode	MeteringMode	37383	9207	SHORT	1
Light source	LightSource	37384	9208	SHORT	1
Flash	Flash	37385	9209	SHORT	1
Lens focal length	FocalLength	37386	920A	RATIONAL	1
Flash energy	FlashEnergy	41483	A20B	RATIONAL	1
Spatial frequency response	SpatialFrequencyResponse	41484	A20C	UNDEFINED	Any
Focal plane X resolution	FocalPlaneXResolution	41486	A20E	RATIONAL	1
Focal plane Y resolution	FocalPlaneYResolution	41487	A20F	RATIONAL	1
Focal plane resolution unit	FocalPlaneResolutionUnit	41488	A210	SHORT	1
Subject location	SubjectLocation	41492	A214	SHORT	2
Exposure index	ExposureIndex	41493	A215	RATIONAL	1
Sensing method	SensingMethod	41495	A217	SHORT	1
File source	FileSource	41728	A300	UNDEFINED	1
Scene type	SceneType	41729	A301	UNDEFINED	1
CFA pattern	CFAPattern	41730	A302	UNDEFINED	Any

A. Tags Relating to Version

ExifVersion

The version of this standard supported. Nonexistence of this field is taken to mean nonconformance to the standard (see section 2.2). Conformance to this standard is indicated by recording "0210" as 4-byte ASCII. Since the type is UNDEFINED, there is no NULL for termination.

Tag	=	36864 (9000.H)
Type	=	UNDEFINED
Count	=	4
Default	=	"0210"

FlashPixVersion

The FlashPix format version supported by a FPXR file. If the FPXR function supports FlashPix format Ver. 1.0, this is indicated similarly to *ExifVersion* by recording "0100" as 4-byte ASCII. Since the type is UNDEFINED, there is no NULL for termination.

Tag	=	40960(A000.H)
Type	=	UNDEFINED
Count	=	4
Default	=	"0100"
0100	=	FlashPix Format Version 1.0
Other	=	reserved

B. Tag Relating to Color Space

ColorSpace

The color space information tag (*ColorSpace*) is always recorded as the color space specifier.

Normally sRGB (=1) is used to define the color space based on the PC monitor conditions and environment. If a color space other than sRGB is used, Uncalibrated (=FFFF.H) is set. Image data recorded as Uncalibrated can be treated as sRGB when it is converted to FlashPix. On sRGB see Appendix E.

Tag = 40961 (A001.H)

Type = SHORT

Count = 1

1 = sRGB

FFFF.H = Uncalibrated

Other = reserved

C. Tags Relating to Image Configuration

PixelXDimension

Information specific to compressed data. When a compressed file is recorded, the valid width of the meaningful image must be recorded in this tag, whether or not there is padding data or a restart marker. This tag should not exist in an uncompressed file. For details see section 2.8.1 and Appendix F.

Tag	= 40962 (A002.H)
Type	= SHORT or LONG
Count	= 1
Default	= none

PixelYDimension

Information specific to compressed data. When a compressed file is recorded, the valid height of the meaningful image must be recorded in this tag, whether or not there is padding data or a restart marker. This tag should not exist in an uncompressed file. For details see section 2.8.1 and Appendix F. Since data padding is unnecessary in the vertical direction, the number of lines recorded in this valid image height tag will in fact be the same as that recorded in the SOF.

Tag	= 40963 (A003.H)
Type	= SHORT or LONG
Count	= 1

ComponentsConfiguration

Information specific to compressed data. The channels of each component are arranged in order from the 1st component to the 4th. For uncompressed data the data arrangement is given in the *PhotometricInterpretation* tag. However, since *PhotometricInterpretation* can only express the order of Y,Cb and Cr, this tag is provided for cases when compressed data uses components other than Y, Cb, and Cr and to enable support of other sequences.

Tag	= 37121 (9101.H)
Type	= UNDEFINED
Count	= 4
Default	= 4 5 6 0 (if RGB uncompressed)
	1 2 3 0 (other cases)
0	= does not exist
1	= Y
2	= Cb
3	= Cr

4	=	R
5	=	G
6	=	B
Other	=	reserved

CompressedBitsPerPixel

Information specific to compressed data. The compression mode used for a compressed image is indicated in unit bits per pixel.

Tag	=	37122 (9102.H)
Type	=	RATIONAL
Count	=	1
Default	=	none

D. Tags Relating to User Information

MakerNote

A tag for manufacturers of Exif writers to record any desired information. The contents are up to the manufacturer.

Tag = 37500 (927C.H)
Type = UNDEFINED
Count = Any
Default = none

UserComment

A tag for Exif users to write keywords or comments on the image besides those in *ImageDescription*, and without the character code limitations of the *ImageDescription* tag.

Tag = 37510 (9286.H)
Type = UNDEFINED
Count = Any
Default = none

The character code used in the *UserComment* tag is identified based on an ID code in a fixed 8-byte area at the start of the tag data area. The unused portion of the area is padded with NULL ("00.H"). ID codes are assigned by means of registration. The designation method and references for each character code are given in Table 6. The value of Count N is determined based on the 8 bytes in the character code area and the number of bytes in the user comment part. Since the TYPE is not ASCII, NULL termination is not necessary (see Fig. 9).

Table 6 Character Codes and their Designation

Character Code	Code Designation (8 Bytes)	References
ASCII	41.H, 53.H, 43.H, 49.H, 49.H, 00.H, 00.H, 00.H	ITU-T T.50 IA5*
JIS	4A.H, 49.H, 53.H, 00.H, 00.H, 00.H, 00.H, 00.H	JIS X0208-1990 ^{xi}
Unicode	55.H, 4E.H, 49.H, 43.H, 4F.H, 44.H, 45.H, 00.H	Unicode Standard ^{xii}
Undefined	00.H, 00.H, 00.H, 00.H, 00.H, 00.H, 00.H, 00.H	Undefined

Exif Private Tag

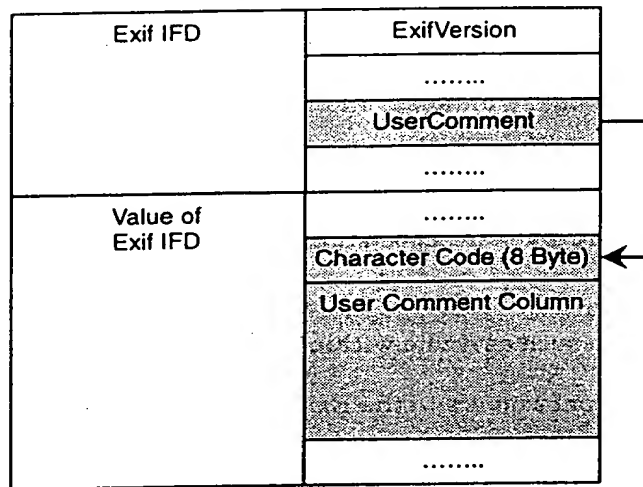


Fig. 9 User Comment Tag

The ID code for the *UserComment* area may be a Defined code such as JIS or ASCII, or may be Undefined. The Undefined name is *UndefinedText*, and the ID code is filled with 8 bytes of all "NULL" ("00.H"). An Exif reader that reads the *UserComment* tag must have a function for determining the ID code. This function is not required in Exif readers that do not use the *UserComment* tag (see Table 7).

Table 7 Implementation of Defined and Undefined Character Codes

ID Code	Exif Reader Implementation
Defined (JIS, ASCII, etc.)	Determines the ID code and displays it in accord with the reader capability.
Undefined (all NULL)	Depends on the localized PC in each country. (If a character code is used for which there is no clear specification like Shift-JIS in Japan, Undefined is used.) Although the possibility of unreadable characters exists, display of these characters is left as a matter of reader implementation.

When a *UserComment* area is set aside, it is recommended that the ID code be ASCII and that the following user comment part be filled with blank characters [20.H].

E. Tag Relating to Related File

RelatedSoundFile

This tag is used to record the name of an audio file related to the image data. The only relational information recorded here is the Exif audio file name and extension (an ASCII string consisting of 8 characters + '.' + 3 characters). The path is not recorded. Stipulations on audio are given in section 3.6.3. File naming conventions are given in section 3.7.1.

When using this tag, audio files must be recorded in conformance to the Exif audio format. Writers are also allowed to store the data such as Audio within APP2 as FlashPix extension stream data.

Audio files must be recorded in conformance to the Exif audio format.

The mapping of Exif image files and audio files is done in any of the three ways shown in Table 8. If multiple files are mapped to one file as in [2] or [3] of this table, the above format is used to record just one audio file name. If there are multiple audio files, the first recorded file is given.

In the case of [3] in Table 8, for example, for the Exif image file "DSC00001.JPG" only "SND00001.WAV" is given as the related Exif audio file.

When there are three Exif audio files "SND00001.WAV", "SND00002.WAV" and "SND00003.WAV", the Exif image file name for each of them, "DSC00001.JPG," is indicated. By combining multiple relational information, a variety of playback possibilities can be supported. The method of using relational information is left to the implementation on the playback side. Since this information is an ASCII character string, it is terminated by NULL.

Table 8 Mapping between Image and Audio Files

	Relationship	Exif Image File	Exif Audio File
[1]	1 to 1	DSC00001.JPG	SND00001.WAV
[2]	Plural to 1	DSC00001.JPG DSC00002.JPG DSC00003.JPG	SND00001.WAV
[3]	1 to plural	DSC00001.JPG	SND00001.WAV SND00002.WAV SND00003.WAV

When this tag is used to map audio files, the relation of the audio file to image data must also be indicated on the audio file end.

Tag = 40964 (A004.H)

Type = ASCII

Count = 13
Default = none

F. Tags Relating to Date and Time

DateTimeOriginal

The date and time when the original image data was generated. For a DSC the date and time the picture was taken are recorded. The format is "YYYY:MM:DD HH:MM:SS" with time shown in 24-hour format, and the date and time separated by one blank character [20.H]. When the date and time are unknown, all the character spaces except colons (":") may be filled with blank characters, or else the Interoperability field may be filled with blank characters. The character string length is 20 bytes including NULL for termination. When the field is left blank, it is treated as unknown.

Tag = 36867 (9003.H)
Type = ASCII
Count = 20
Default = none

DateTimeDigitized

The date and time when the image was stored as digital data. If, for example, an image was captured by DSC and at the same time the file was recorded, then the *DateTimeOriginal* and *DateTimeDigitized* will have the same contents. The format is "YYYY:MM:DD HH:MM:SS" with time shown in 24-hour format, and the date and time separated by one blank character [20.H]. When the date and time are unknown, all the character spaces except colons (":") may be filled with blank characters, or else the Interoperability field may be filled with blank characters. The character string length is 20 bytes including NULL for termination. When the field is left blank, it is treated as unknown.

Tag = 36868 (9004.H)
Type = ASCII
Count = 20
Default = none

SubsecTime

A tag used to record fractions of seconds for the *DateTime* tag.

Tag = 37520 (9290.H)
Type = ASCII
Count = Any
Default = none

SubsecTimeOriginal

A tag used to record fractions of seconds for the *DateTimeOriginal* tag.

Tag = 37521 (9291.H)
Type = ASCII
N = Any
Default = none

SubsecTimeDigitized

A tag used to record fractions of seconds for the *DateTimeDigitized* tag.

Tag = 37522 (9292.H)
Type = ASCII
N = Any
Default = none

Note: Recording subsecond data (*SubsecTime*, *SubsecTimeOriginal*, *SubsecTimeDigitized*)

The tag type is ASCII and the string length including NULL is variable length. When the number of valid digits is up to the second decimal place, the subsecond value goes in the Value position. When it is up to four decimal places, an address value is Interoperability, with the subsecond value put in the location pointed to by that address. (Since the count of ASCII type field Interoperability is a value that includes NULL, when the number of valid digits is up to four decimal places the count is 5, and the offset value goes in the Value Offset field. See section 2.6.2.) Note that the subsecond tag differs from the *DateTime* tag and other such tags already defined in TIFF Rev. 6.0, and that both are recorded in the Exif IFD.

Ex.: September 9, 1998, 9:15:30.130

(the number of valid digits is up to the third decimal place)

DateTime 1996:09:01 09:15:30 [NULL]

SubSecTime 130 [NULL]

If the string length is longer than the number of valid digits, the digits are aligned with the start of the area and the rest is filled with blank characters [20.H]. If the subsecond data is unknown, the Interoperability area can be filled with blank characters.

Examples when subsecond data is 0.130 seconds:

Ex. 1) '1','3','0',[NULL]

Ex. 2) '1','3','0',[20.H],[NULL]

Ex. 3) '1','3','0',[20.H],[20.H],[20.H],[20.H],[20.H],[NULL]

Example when subsecond data is unknown:

Ex. 4) {20.H}, {20.H}, {20.H}, {20.H}, {20.H}, {20.H}, {20.H}, {20.H}, [NULL]

G. Tags Relating to Picture-Taking Conditions

ExposureTime

Exposure time, given in seconds (sec).

Tag = 33434 (829A.H)
Type = RATIONAL
Count = 1
Default = none

ShutterSpeedValue

Shutter speed. The unit is the APEX (Additive System of Photographic Exposure) setting (see Appendix C).

Tag = 37377 (9201.H)
Type = SRATIONAL
Count = 1
Default = none

ApertureValue

The lens aperture. The unit is the APEX value.

Tag = 37378 (9202.H)
Type = RATIONAL
Count = 1
Default = none

BrightnessValue

The value of brightness. The unit is the APEX value. Ordinarily it is given in the range of -99.99 to 99.99.

Tag = 37379 (9203.H)
Type = SRATIONAL
Count = 1
Default = none

ExposureBiasValue

The exposure bias. The unit is the APEX value. Ordinarily it is given in the range of -99.99 to 99.99.

Tag = 37380 (9204.H)
Type = SRATIONAL

Count = 1
Default = none

MaxApertureValue

The smallest F number of the lens. The unit is the APEX value. Ordinarily it is given in the range of 00.00 to 99.99, but it is not limited to this range.

Tag = 37381 (9205.H)
Type = RATIONAL
Count = 1
Default = none

SubjectDistance

The distance to the subject, given in meters.

Tag = 37382 (9206.H)
Type = RATIONAL
Count = 1
Default = none

MeteringMode

The metering mode.

Tag = 37383 (9207.H)
Type = SHORT
Count = 1
Default = 0

0	= unknown
1	= Average
2	= CenterWeightedAverage
3	= Spot
4	= MultiSpot
5	= Pattern
6	= Partial
7 to 254	= reserved
255	= other

LightSource

The kind of light source.

Tag = 37384 (9208.H)

Type	=	SHORT
Count	=	1
Default	=	0
0	=	unknown
1	=	Daylight
2	=	Fluorescent
3	=	Tungsten
17	=	Standard light A
18	=	Standard light B
19	=	Standard light C
20	=	D55
21	=	D65
22	=	D75
23 to 254	=	reserved
255	=	other

Flash

This tag is recorded when an image is taken using a strobe light (flash). Bit 0 indicates the flash firing status, and bits 1 and 2 indicate the flash return status (see Fig. 10).

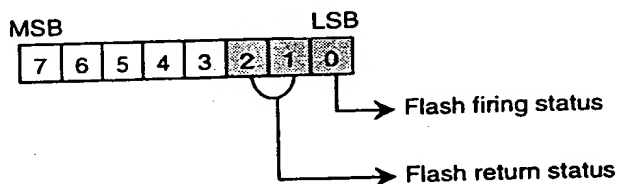


Fig. 10 Bit Coding of the Flash Tag

Tag	=	37385 (9209.H)
Type	=	SHORT
Count	=	1

Values for bit 0 indicating whether the flash fired.

0b	=	Flash did not fire.
1b	=	Flash fired.

Values for bits 1 and 2 indicating the status of returned light.

00b	=	No strobe return detection function
01b	=	reserved
10b	=	Strobe return light not detected.
11b	=	Strobe return light detected.

Resulting Flash tag values.

0000.H = Flash did not fire.
0001.H = Flash fired.
0005.H = Strobe return light not detected.
0007.H = Strobe return light detected.
Other = reserved

FocalLength

The actual focal length of the lens, in mm. Conversion is not made to the focal length of a 35 mm film camera.

Tag = 37386 (920A.H)
Type = RATIONAL
Count = 1
Default = none

FNumber

The F number.

Tag = 33437 (829D.H)
Type = RATIONAL
Count = 1
Default = none

ExposureProgram

The class of the program used by the camera to set exposure when the picture is taken. The tag values are as follows.

Tag = 34850 (8822.H)
Type = SHORT
Count = 1
Default = 0

0 = Not defined
1 = Manual
2 = Normal program
3 = Aperture priority
4 = Shutter priority
5 = Creative program (biased toward depth of field)
6 = Action program (biased toward fast shutter speed)
7 = Portrait mode (for closeup photos with the background out of focus)

- 8 = Landscape mode (for landscape photos with the background in focus)
 9 to 255 = reserved

SpectralSensitivity

Indicates the spectral sensitivity of each channel of the camera used. The tag value is an ASCII string compatible with the standard^{xiii} developed by the ASTM Technical committee.

Tag = 34852 (8824.H)
 Type = ASCII
 Count = Any
 Default = none

ISOSpeedRatings

Indicates the ISO Speed and ISO Latitude of the camera or input device as specified in ISO 12232^{xiv}.

Tag = 34855 (8827.H)
 Type = SHORT
 Count = Any
 Default = none

OECF

Indicates the Opto-Electronic Conversion Function (OECF) specified in ISO 14524^{xv}. *OECF* is the relationship between the camera optical input and the image values.

Tag = 34856 (8828.H)
 Type = UNDEFINED
 Count = ANY
 Default = none

When this tag records an *OECF* of *m* rows and *n* columns, the values are as in Fig. 11.

Length	Type	Meaning
2	SHORT	Columns = <i>n</i>
2	SHORT	Rows = <i>m</i>
Any	ASCII	0th column item name (NULL terminated)
:	:	:
Any	ASCII	<i>n</i> -1th column item name (NULL terminated)
8	SRATIONAL	OECF value [0,0]
:	:	:
8	SRATIONAL	OECF value [<i>n</i> -1,0]
8	SRATIONAL	OECF value [0, <i>m</i> -1]
:	:	:
8	SRATIONAL	OECF value [<i>n</i> -1, <i>m</i> -1]

Fig. 11 OECF Description

Table 9 gives a simple example.

Table 9 Example of Exposure and RGB Output Level

Camera log Aperture	R Output Level	G Output Level	B Output Level
-3.0	10.2	12.4	8.9
-2.0	48.1	47.5	48.3
-1.0	150.2	152.0	149.8

FlashEnergy

Indicates the strobe energy at the time the image is captured, as measured in Beam Candle Power Seconds (BCPS).

Tag = 41483 (A20B.H)
 Type = RATIONAL
 Count = 1
 Default = none

SpatialFrequencyResponse

This tag records the camera or input device spatial frequency table and SFR values in the direction of image width, image height, and diagonal direction, as specified in ISO 12233^{rev}.

Tag = 41484 (A20CH)
 Type = UNDEFINED
 Count = ANY
 Default = none

When the spatial frequency response for m rows and n columns is recorded, the values are as shown in Fig. 12.

Length	Type	Meaning
2	SHORT	Columns = n
2	SHORT	Rows = m
Any	ASCII	0th column item name (NULL terminated)
:	:	:
Any	ASCII	$n-1$ th column item name (NULL terminated)
8	RATIONAL	SFR value [0,0]
:	:	:
8	RATIONAL	SFR value [$n-1,0$]
8	RATIONAL	SFR value [0, $m-1$]
:	:	:
8	RATIONAL	SFR value [$n-1,m-1$]

Fig. 12 Spatial Frequency Response Description

Table 10 gives a simple example.

Table 10 Example of Spatial Frequency Response

Spatial Frequency (lw/ph)	Along Image Width	Along Image Height
0.1	1.00	1.00
0.2	0.90	0.95
0.3	0.80	0.85

FocalPlaneXResolution

Indicates the number of pixels in the image width (X) direction per *FocalPlaneResolutionUnit* on the camera focal plane.

Tag = 41486 (A20E.H)
 Type = RATIONAL
 Count = 1
 Default = none

FocalPlaneYResolution

Indicates the number of pixels in the image height (Y) direction per *FocalPlaneResolutionUnit* on the camera focal plane.

Tag = 41487 (A20F.H)
 Type = RATIONAL
 Count = 1
 Default = none

FocalPlaneResolutionUnit

Indicates the unit for measuring *FocalPlaneXResolution* and *FocalPlaneYResolution*. This value is the same as the *ResolutionUnit*.

Tag = 41488 (A210.H)
 Type = SHORT
 Count = 1
 Default = 2 (inch)

Note on use of tags concerning focal plane resolution

These tags record the actual focal plane resolutions of the main image which is written as a file after processing instead of the pixel resolution of the image sensor in the camera. It should be noted carefully that the data from the image sensor is resampled.

These tags are used at the same time as a *FocalLength* tag when the angle of field of the recorded image is to be calculated precisely.

SubjectLocation

Indicates the location of the main subject in the scene. The value of this tag represents the pixel at the center of the main subject relative to the left edge, prior to rotation processing as per the *Rotation* tag. The first value indicates the X column number and second indicates the Y row number.

Tag = 41492 (A214.H)
Type = SHORT
Count = 2
Default = none

ExposureIndex

Indicates the exposure index selected on the camera or input device at the time the image is captured.

Tag = 41493 (A215.H)
Type = RATIONAL
Count = 1
Default = none

SensingMethod

Indicates the image sensor type on the camera or input device. The values are as follows.

Tag = 41495 (A217.H)
Type = SHORT
Count = 1
Default = none

1	= Not defined
2	= One-chip color area sensor
3	= Two-chip color area sensor
4	= Three-chip color area sensor
5	= Color sequential area sensor
7	= Trilinear sensor
8	= Color sequential linear sensor
Other	= reserved

FileSource

Indicates the image source. If a DSC recorded the image, this tag value of this tag always be set to 3, indicating that the image was recorded on a DSC.

Tag = 41728 (A300.H)
Type = UNDEFINED
Count = 1

Default = 3
 3 = DSC
 Other = reserved

SceneType

Indicates the type of scene. If a DSC recorded the image, this tag value must always be set to 1, indicating that the image was directly photographed.

Tag = 41729 (A301.H)
 Type = UNDEFINED
 Count = 1
 Default = 1
 1 = A directly photographed image
 Other = reserved

CFAPattern

Indicates the color filter array (CFA) geometric pattern of the image sensor when a one-chip color area sensor is used. It does not apply to all sensing methods.

Tag = 41730 (A302.H)
 Type = UNDEFINED
 Count = ANY

Fig. 13 shows how a CFA pattern is recorded for a one-chip color area sensor when the color filter array is repeated in $m \times n$ (vertical \times lateral) pixel units.

Length	Type	Meaning
2	SHORT	Horizontal repeat pixel unit = n
2	SHORT	Vertical repeat pixel unit = m
1	BYTE	CFA value [0.0]
:	:	:
1	BYTE	CFA value [$n-1.0$]
1	BYTE	CFA value [0. $m-1$]
:	:	:
1	BYTE	CFA value [$n-1.m-1$]

Fig. 13 CFA Pattern Description

The relation of color filter color to CFA value is shown in Table 11.

Table 11 Color Filter Color and CFA Value

Filter Color	CFA Value
RED	00.H
GREEN	01.H
BLUE	02.H
CYAN	03.H
MAGENTA	04.H
YELLOW	05.H
WHITE	06.H

For example, when the CFA pattern values are {0002.H, 0002.H, 01.H, 00.H, 02.H, 01.H}, the color filter array is as shown in Fig. 14.

G	R	G	R
B	G	B	G
G	R	G	R
B	G	B	G
:	:	:	:	

Fig. 14 Color Filter Array

2.6.6. GPS Attribute Information

The attribute information (field names and codes) recorded in the GPS Info IFD is given in Table 12, followed by an explanation of the contents.

Table 12 GPS Attribute Information

Tag Name	Field Name	Tag ID		Type	Count
		Dec	Hex		
A. Tags Relating to GPS					
GPS tag version	GPSTagVersionID	0	0	BYTE	4
North or South Latitude	GPSTagLatitudeRef	1	1	ASCII	2
Latitude	GPSTagLatitude	2	2	RATIONAL	3
East or West Longitude	GPSTagLongitudeRef	3	3	ASCII	2
Longitude	GPSTagLongitude	4	4	RATIONAL	3
Altitude reference	GPSTagAltitudeRef	5	5	BYTE	1
Altitude	GPSTagAltitude	6	6	RATIONAL	1
GPS time (atomic clock)	GPSTimeStamp	7	7	RATIONAL	3
GPS satellites used for measurement	GPSSatellites	8	8	ASCII	Any
GPS receiver status	GPSStatus	9	9	ASCII	2
GPS measurement mode	GPSMeasureMode	10	A	ASCII	2
Measurement precision	GPSDOP	11	B	RATIONAL	1
Speed unit	GPSSpeedRef	12	C	ASCII	2
Speed of GPS receiver	GPSSpeed	13	D	RATIONAL	1
Reference for direction of movement	GPSTrackRef	14	E	ASCII	2
Direction of movement	GPSTrack	15	F	RATIONAL	1
Reference for direction of image	GPSTagImgDirectionRef	16	10	ASCII	2
Direction of image	GPSTagImgDirection	17	11	RATIONAL	1
Geodetic survey data used	GPSTagMapDatum	18	12	ASCII	Any
Reference for latitude of destination	GPSTagDestLatitudeRef	19	13	ASCII	2
Latitude of destination	GPSTagDestLatitude	20	14	RATIONAL	3
Reference for longitude of destination	GPSTagDestLongitudeRef	21	15	ASCII	2
Longitude of destination	GPSTagDestLongitude	22	16	RATIONAL	3
Reference for bearing of destination	GPSTagDestBearingRef	23	17	ASCII	2
Bearing of destination	GPSTagDestBearing	24	18	RATIONAL	1
Reference for distance to destination	GPSTagDestDistanceRef	25	19	ASCII	2
Distance to destination	GPSTagDestDistance	26	1A	RATIONAL	1

A. Tags Relating to GPS

GPSTagVersionID

Indicates the version of *GPSTagInfoFD*. The version is given as 2.0.0.0. This tag is mandatory when *GPSTagInfo* tag is present. (Note: The *GPSTagVersionID* tag is given in bytes, unlike the *ExifVersion* tag. When the version is 2.0.0.0, the tag value is 02000000.H.)

Tag = 0 (0.H)
Type = BYTE
Count = 4
Default = 2.0.0.0
2.0.0.0 = Version 2.0
Other = reserved

GPSTagLatitudeRef

Indicates whether the latitude is north or south latitude. The ASCII value 'N' indicates north latitude, and 'S' is south latitude.

Tag = 1 (1.H)
Type = ASCII
Count = 2
Default = none
'N' = North latitude
'S' = South latitude
Other = reserved

GPSTagLatitude

Indicates the latitude. The latitude is expressed as three RATIONAL values giving the degrees, minutes, and seconds, respectively. When degrees, minutes and seconds are expressed, the format is dd/1,mm/1,ss/1. When degrees and minutes are used and, for example, fractions of minutes are given up to two decimal places, the format is dd/1,mmmm/100,0/1.

Tag = 2 (2.H)
Type = RATIONAL
Count = 3
Default = none

GPSTagLongitudeRef

Indicates whether the longitude is east or west longitude. ASCII 'E' indicates east longitude, and 'W'

is west longitude.

Tag	=	3 (3.H)
Type	=	ASCII
Count	=	2
Default	=	none
'E'	=	East longitude
'W'	=	West longitude
Other	=	reserved

GPSLongitude

Indicates the longitude. The longitude is expressed as three RATIONAL values giving the degrees, minutes, and seconds, respectively. When degrees, minutes and seconds are expressed, the format is ddd/1,mm/1,ss/1. When degrees and minutes are used and, for example, fractions of minutes are given up to two decimal places, the format is ddd/1,mmmm/100,0/1.

Tag	=	4 (4.H)
Type	=	RATIONAL
Count	=	3
Default	=	none

GPSAltitudeRef

Indicates the altitude used as the reference altitude. In this version the reference altitude is sea level, so this tag must be set to 0. The reference unit is meters. Note that this tag is BYTE type, unlike other reference tags.

Tag	=	5 (5.H)
Type	=	BYTE
Count	=	1
Default	=	0
0	=	Sea level
Other	=	reserved

GPSAltitude

Indicates the altitude based on the reference in *GPSAltitudeRef*. Altitude is expressed as one RATIONAL value. The reference unit is meters.

Tag	=	6 (6.H)
Type	=	RATIONAL
Count	=	1
Default	=	none

GPSTimeStamp

Indicates the time as UTC (Coordinated Universal Time). TimeStamp is expressed as three RATIONAL values giving the hour, minute, and second.

Tag = 7 (7.H)
Type = RATIONAL
Count = 3
Default = none

GPSSatellites

Indicates the GPS satellites used for measurements. This tag can be used to describe the number of satellites, their ID number, angle of elevation, azimuth, SNR and other information in ASCII notation. The format is not specified. If the GPS receiver is incapable of taking measurements, value of the tag must be set to NULL.

Tag = 8 (8.H)
Type = ASCII
Count = Any
Default = none

GPSStatus

Indicates the status of the GPS receiver when the image is recorded. 'A' means measurement is in progress, and 'V' means the measurement is Interoperability.

Tag = 9 (9.H)
Type = ASCII
Count = 2
Default = none
'A' = Measurement in progress
'V' = Measurement Interoperability
Other = reserved

GPSMeasureMode

Indicates the GPS measurement mode. '2' means two-dimensional measurement and '3' means three-dimensional measurement is in progress.

Tag = 10 (A.H)
Type = ASCII
Count = 2
Default = none
'2' = 2-dimensional measurement

'3' = 3-dimensional measurement
Other = reserved

GPSDOP

Indicates the GPS DOP (data degree of precision). An HDOP value is written during two-dimensional measurement, and PDOP during three-dimensional measurement.

Tag = 11 (B.H)
Type = RATIONAL
Count = 1
Default = none

GPSSpeedRef

Indicates the unit used to express the GPS receiver speed of movement. 'K' 'M' and 'N' represents kilometers per hour, miles per hour, and knots.

Tag = 12 (C.H)
Type = ASCII
Count = 2
Default = 'K'
'K' = Kilometers per hour
'M' = Miles per hour
'N' = Knots
Other = reserved

GPSSpeed

Indicates the speed of GPS receiver movement.

Tag = 13 (D.H)
Type = RATIONAL
Count = 1
Default = none

GPSTrackRef

Indicates the reference for giving the direction of GPS receiver movement. 'T' denotes true direction and 'M' is magnetic direction.

Tag = 14 (E.H)
Type = ASCII
Count = 2
Default = 'T'

'T' = True direction
 'M' = Magnetic direction
 Other = reserved

GPSTrack

Indicates the direction of GPS receiver movement. The range of values is from 0.00 to 359.99.

Tag = 15 (F.H)
 Type = RATIONAL
 Count = 1
 Default = none

GPSImgDirectionRef

Indicates the reference for giving the direction of the image when it is captured. 'T' denotes true direction and 'M' is magnetic direction.

Tag = 16 (10.H)
 Type = ASCII
 Count = 2
 Default = 'T'
 'T' = True direction
 'M' = Magnetic direction
 Other = reserved

GPSImgDirection

Indicates the direction of the image when it was captured. The range of values is from 0.00 to 359.99.

Tag = 17 (11.H)
 Type = RATIONAL
 Count = 1
 Default = none

GPSMapDatum

Indicates the geodetic survey data used by the GPS receiver. If the survey data is restricted to Japan, the value of this tag is 'TOKYO' or 'WGS-84'. If a *GPS Info* tag is recorded, it is strongly recommended that this tag be recorded.

Tag = 18 (12.H)
 Type = ASCII
 Count = Any
 Default = none

GPSTDestLatitudeRef

Indicates whether the latitude of the destination point is north or south latitude. The ASCII value 'N' indicates north latitude, and 'S' is south latitude.

Tag = 19 (13.H)
Type = ASCII
Count = 2
Default = none
'N' = North latitude
'S' = South latitude
Other = reserved

GPSTDestLatitude

Indicates the latitude of the destination point. The latitude is expressed as three RATIONAL values giving the degrees, minutes, and seconds, respectively. When degrees, minutes and seconds are expressed, the format is dd/1,mm/1,ss/1. When degrees and minutes are used and, for example, fractions of minutes are given up to two decimal places, the format is dd/1,mmmm/100,0/1.

Tag = 20 (14.H)
Type = RATIONAL
Count = 3
Default = none

GPSTDestLongitudeRef

Indicates whether the longitude of the destination point is east or west longitude. ASCII 'E' indicates east longitude, and 'W' is west longitude.

Tag = 21 (15.H)
Type = ASCII
Count = 2
Default = none
'E' = East longitude
'W' = West longitude
Other = reserved

GPSTDestLongitude

Indicates the longitude of the destination point. The longitude is expressed as three RATIONAL values giving the degrees, minutes, and seconds, respectively. When degrees, minutes and seconds are expressed, the format is ddd/1,mm/1,ss/1. When degrees and minutes are used and, for example, fractions of minutes are given up to two decimal places, the format is ddd/1,mmmm/100,0/1.

Tag = 22 (16.H)
Type = RATIONAL
Count = 3
Default = none

GPSTDestBearingRef

- Indicates the reference used for giving the bearing to the destination point. 'T' denotes true direction and 'M' is magnetic direction.

Tag = 23 (17.H)
Type = ASCII
Count = 2
Default = 'T'
 'T' = True direction
 'M' = Magnetic direction
 Other = reserved

GPSTDestBearing

Indicates the bearing to the destination point. The range of values is from 0.00 to 359.99.

Tag = 24 (18.H)
Type = RATIONAL
Count = 1
Default = none

GPSTDestDistanceRef

Indicates the unit used to express the distance to the destination point. 'K', 'M' and 'N' represent kilometers, miles and knots.

Tag = 25 (19.H)
Type = ASCII
Count = 2
Default = 'K'
 'K' = Kilometers
 'M' = Miles
 'N' = Knots
 Other = reserved

GPSTDestDistance

Indicates the distance to the destination point.

Tag = 26 (1A.H)
Type = RATIONAL
Count = 1
Default = none

Note: When the tag Type is ASCII, it must be terminated with NULL.

It must be noted carefully that since the value count includes the terminator NULL, the total count is the number of data+1. For example, *GPSLatitudeRef* cannot have any values other than Type ASCII 'N' or 'S'; but because the terminator NULL is added, the value of N is 2.

2.6.7. Interoperability IFD Attribute Information

The attached information(field name, code) stored in Interoperability IFD is listed in Table 12-2 and the meaning will be explained also.

Table 13 Interoperability IFD Attribute Information

Tag Name	Field Name	Tag ID		Type	Count
		Dec	Hex		
A. Attached Information Related to Interoperability		0	0	ASCII	Any
	Interoperability Identification Interoperability Index				

A. Tags Relating to Interoperability

The rules for Exif image files defines the description of the following tag. Other tags stored in Interoperability IFD may be defined dependently to each Interoperability rule.

Interoperability Index

Indicates the identification of the Interoperability rule.

Use "R98" for stating ExifR98 Rules when using intreoperability rules recommended in Appendix D. Four bytes used including the termination code(NULL). See the separate volume of Recommended Exif Interoperability Rules (ExifR98) for other tags used for ExifR98.

Tag = 1 (1.H)

Type = ASCII

Count = Any

Default = none

"R98" = Recommended Exif Interoperability Rules (ExifR98)

2.6.8. Tag Support Levels

The tags and their support levels are given here.

A. Primary Image (0th IFD) Support Levels

The support levels of primary image (0th IFD) tags are given in Table 14, Table 15, Table 16 and Table 17.

Table 14 Tag Support Levels (1) - 0th IFD TIFF Tags -

Tag Name	Field Name	Tag ID		Uncompressed			Compressed
		Dec	Hex	Chunky	Planar	YCC	
Image width	ImageWidth	256	100	M	M	M	J
Image height	ImageLength	257	101	M	M	M	J
Number of bits per component	BitsPerSample	258	102	M	M	M	J
Compression scheme	Compression	259	103	M	M	M	J
Pixel composition	PhotometricInterpretation	262	106	M	M	M	J
Image title	ImageDescription	270	10E	R	R	R	R
Manufacturer of image input equipment	Make	271	10F	R	R	R	R
Model of image input equipment	Model	272	110	R	R	R	R
Image data location	StripOffsets	273	111	M	M	M	N
Orientation of image	Orientation	274	112	R	R	R	R
Number of components	SamplesPerPixel	277	115	M	M	M	J
Number of rows per strip	RowsPerStrip	278	116	M	M	M	N
Bytes per compressed strip	StripByteCounts	279	117	M	M	M	N
Image resolution in width direction	XResolution	282	11A	M	M	M	M
Image resolution in height direction	YResolution	283	11B	M	M	M	M
Image data arrangement	PlanarConfiguration	284	11C	O	M	O	J
Unit of X and Y resolution	ResolutionUnit	296	128	M	M	M	M
Transfer function	TransferFunction	301	12D	R	R	R	R
Software used	Software	305	131	O	O	O	O
File change date and time	DateTime	306	132	R	R	R	R
Person who created the image	Artist	315	13B	O	O	O	O
White point chromaticity	WhitePoint	318	13E	O	O	O	O
Chromaticities of primaries	PrimaryChromaticities	319	13F	O	O	O	O
Offset to JPEG SOI	JPEGInterchangeFormat	513	201	N	N	N	N
Bytes of JPEG data	JPEGInterchangeFormatLength	514	202	N	N	N	N
Color space transformation matrix coefficients	YCbCrCoefficients	529	211	N	N	O	O
Subsampling ratio of Y to C	YCbCrSubSampling	530	212	N	N	M	J
Y and C positioning	YCbCrPositioning	531	213	N	N	M	M
Pair of black and white reference values	ReferenceBlackWhite	532	214	O	O	O	O
Copyright holder	Copyright	33432	8298	O	O	O	O
Exif tag	Exif IFD Pointer	34665	8769	M	M	M	M
GPS tag	GPSInfo IFD Pointer	34853	8825	O	O	O	O

Notation

- M : Mandatory (must be recorded)
- R : Conditionally mandatory (must be recorded if hardware permits)
- O : Optional
- N : Not recorded
- J : Included in JPEG marker and so not recorded

Table 15 Tag Support Levels (2) - 0th IFD Exif Private Tags -

Tag Name	Field Name	Tag ID		Uncompressed			Compressed
		Dec	Hex	Chunky	Planar	YCC	
Exposure time	ExposureTime	33434	829A	O	O	O	O
F number	FNumber	33437	829D	O	O	O	O
Exposure program	ExposureProgram	34850	8822	O	O	O	O
Spectral sensitivity	SpectralSensitivity	34852	8824	O	O	O	O
ISO speed ratings	ISOSpeedRatings	34855	8827	O	O	O	O
Optoelectric coefficient	OECF	34856	8828	O	O	O	O
Exif Version	ExifVersion	36864	9000	M	M	M	M
Date and time original image was generated	DateTimeOriginal	36867	9003	O	O	O	O
Date and time image was made digital data	DateTimeDigitized	36868	9004	O	O	O	O
Meaning of each component	ComponentsConfiguration	37121	9101	N	N	N	M
Image compression mode	CompressedBitsPerPixel	37122	9102	N	N	N	O
Shutter speed	ShutterSpeedValue	37377	9201	O	O	O	O
Aperture	ApertureValue	37378	9202	O	O	O	O
Brightness	BrightnessValue	37379	9203	O	O	O	O
Exposure bias	ExposureBiasValue	37380	9204	O	O	O	O
Maximum lens aperture	MaxApertureValue	37381	9205	O	O	O	O
Subject distance	SubjectDistance	37382	9206	O	O	O	O
Metering mode	MeteringMode	37383	9207	O	O	O	O
Light source	LightSource	37384	9208	O	O	O	O
Flash	Flash	37385	9209	O	O	O	O
Lens focal length	FocalLength	37386	920A	O	O	O	O
Manufacturer notes	MakerNote	37500	927C	O	O	O	O
User comments	UserComment	37510	9286	O	O	O	O
DateTime subseconds	SubSecTime	37520	9290	O	O	O	O
DateTimeOriginal subseconds	SubSecTimeOriginal	37521	9291	O	O	O	O
DateTimeDigitized subseconds	SubSecTimeDigitized	37522	9292	O	O	O	O
Supported FlashPix version	FlashPixVersion	40960	A000	M	M	M	M
Color space information	ColorSpace	40961	A001	M	M	M	M
Valid image width	PixelXDimension	40962	A002	N	N	N	M
Valid image height	PixelYDimension	40963	A003	N	N	N	M
Related audio file	RelatedSoundFile	40964	A004	O	O	O	O
Interoperability tag	Interoperability IFD Pointer	40965	A005	N	N	N	O
Flash energy	FlashEnergy	41483	A20B	O	O	O	O
Spatial frequency response	SpatialFrequencyResponse	41484	A20C	O	O	O	O
Focal plane X resolution	FocalPlaneXResolution	41486	A20E	O	O	O	O
Focal plane Y resolution	FocalPlaneYResolution	41487	A20F	O	O	O	O
Focal plane resolution unit	FocalPlaneResolutionUnit	41488	A210	O	O	O	O
Subject location	SubjectLocation	41492	A214	O	O	O	O
Exposure index	ExposureIndex	41493	A215	O	O	O	O
Sensing method	SensingMethod	41495	A217	O	O	O	O
File source	FileSource	41728	A300	O	O	O	O
Scene type	SceneType	41729	A301	O	O	O	O
CFA pattern	CFAPattern	41730	A302	O	O	O	O

Notation

- M : Mandatory (must be recorded)
- R : Conditionally mandatory (must be recorded if hardware permits)
- O : Optional
- N : Not recorded
- J : Included in JPEG marker and so not recorded

Table 16 Tag Support Levels (3) - 0th IFD GPS Info Tags -

Tag Name	Field Name	Tag ID		Uncompressed			Comp- ressed
		Dec	Hex	Chunky	Planar	YCC	
GPS tag version	GPSTagVersionID	0	0	O	O	O	O
North or South Latitude	GPSTLatitudeRef	1	1	O	O	O	O
Latitude	GPSTLatitude	2	2	O	O	O	O
East or West Longitude	GPSTLongitudeRef	3	3	O	O	O	O
Longitude	GPSTLongitude	4	4	O	O	O	O
Altitude reference	GPSTAltitudeRef	5	5	O	O	O	O
Altitude	GPSTAltitude	6	6	O	O	O	O
GPS time (atomic clock)	GPSTimeStamp	7	7	O	O	O	O
GPS satellites used for measurement	GPSSatellites	8	8	O	O	O	O
GPS receiver status	GPSTStatus	9	9	O	O	O	O
GPS measurement mode	GPSTMeasureMode	10	A	O	O	O	O
Measurement precision	GPSTDOP	11	B	O	O	O	O
Speed unit	GPSTSpeedRef	12	C	O	O	O	O
Speed of GPS receiver	GPSTSpeed	13	D	O	O	O	O
Reference for direction of movement	GPSTTrackRef	14	E	O	O	O	O
Direction of movement	GPSTTrack	15	F	O	O	O	O
Reference for direction of image	GPSTImgDirectionRef	16	10	O	O	O	O
Direction of image	GPSTImgDirection	17	11	O	O	O	O
Geodetic survey data used	GPSTMapDatum	18	12	O	O	O	O
Reference for latitude of destination	GPSTDestLatitudeRef	19	13	O	O	O	O
Latitude of destination	GPSTDestLatitude	20	14	O	O	O	O
Reference for longitude of destination	GPSTDestLongitudeRef	21	15	O	O	O	O
Longitude of destination	GPSTDestLongitude	22	16	O	O	O	O
Reference for bearing of destination	GPSTDestBearingRef	23	17	O	O	O	O
Bearing of destination	GPSTDestBearing	24	18	O	O	O	O
Reference for distance to destination	GPSTDestDistanceRef	25	19	O	O	O	O
Distance to destination	GPSTDestDistance	26	1A	O	O	O	O

Table 17 Tag Support Levels (4) - 0th IFD Interoperability Tag -

Tag Name	Field Name	Tag ID		Uncompressed			Comp- ressed
		Dec	Hex	Chunky	Planar	YCC	
Interoperability Identification	Interoperability Index	0	0	N	N	N	O

Notation

M : Mandatory (must be recorded)

R : Conditionally mandatory (must be recorded if hardware permits)

O : Optional

N : Not recorded

J : Included in JPEG marker and so not recorded

B. Thumbnail (1st IFD) Support Levels

The support levels of thumbnail (1st IFD) tags are shown in Table 18.

Table 18 Tag Support Levels (5) - 1st IFD TIFF Tag -

Tag Name	Field Name	Tag ID		Uncompressed			Compressed
		Dec	Hex	Chunky	Planar	YCC	
Image width	ImageWidth	256	100	M	M	M	J
Image height	ImageLength	257	101	M	M	M	J
Number of bits per component	BitsPerSample	258	102	M	M	M	J
Compression scheme	Compression	259	103	M	M	M	M
Pixel composition	PhotometricInterpretation	262	106	M	M	M	J
Image title	ImageDescription	270	10E	O	O	O	O
Manufacturer of image input equipment	Make	271	10F	O	O	O	O
Model of image input equipment	Model	272	110	O	O	O	O
Image data location	StripOffsets	273	111	M	M	M	N
Orientation of image	Orientation	274	112	O	O	O	O
Number of components	SamplesPerPixel	277	115	M	M	M	J
Number of rows per strip	RowsPerStrip	278	116	M	M	M	N
Bytes per compressed strip	StripByteCounts	279	117	M	M	M	N
Image resolution in width direction	XResolution	282	11A	M	M	M	M
Image resolution in height direction	YResolution	283	11B	M	M	M	M
Image data arrangement	PlanarConfiguration	284	11C	O	M	O	J
Unit of X and Y resolution	ResolutionUnit	296	128	M	M	M	M
Transfer function	TransferFunction	301	12D	O	O	O	O
Software used	Software	305	131	O	O	O	O
File change date and time	DateTime	306	132	O	O	O	O
Person who created the image	Artist	315	13B	O	O	O	O
White point chromaticity	WhitePoint	318	13E	O	O	O	O
Chromaticities of primaries	PrimaryChromaticities	319	13F	O	O	O	O
Offset to JPEG SOI	JPEGInterchangeFormat	513	201	N	N	N	M
Bytes of JPEG data	JPEGInterchangeFormatLength	514	202	N	N	N	M
Color space transformation matrix coefficients	YCbCrCoefficients	529	211	N	N	O	O
Subsampling ratio of Y to C	YCbCrSubSampling	530	212	N	N	M	J
Y and C positioning	YCbCrPositioning	531	213	N	N	O	O
Pair of black and white reference values	ReferenceBlackWhite	532	214	O	O	O	O
Copyright holder	Copyright	33432	8298	O	O	O	O
Exif tag	Exif IFD Pointer	34665	8769	O	O	O	O
GPS tag	GPSInfo IFD Pointer	34853	8825	O	O	O	O

Notation

- M : Mandatory (must be recorded)
- R : Conditionally mandatory (must be recorded if hardware permits)
- O : Optional
- N : Not recorded
- J : Included in JPEG marker and so not recorded

Network Working Group
Request for Comments: 1945
Category: Informational

T. Berners-Lee
MIT/LCS
R. Fielding
UC Irvine
H. Frystyk
MIT/LCS
May 1996

Hypertext Transfer Protocol -- HTTP/1.0

Status of This Memo

This memo provides information for the Internet community. This memo does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

IESG Note:

The IESG has concerns about this protocol, and expects this document to be replaced relatively soon by a standards track document.

Abstract

The Hypertext Transfer Protocol (HTTP) is an application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems. It is a generic, stateless, object-oriented protocol which can be used for many tasks, such as name servers and distributed object management systems, through extension of its request methods (commands). A feature of HTTP is the typing of data representation, allowing systems to be built independently of the data being transferred.

HTTP has been in use by the World-Wide Web global information initiative since 1990. This specification reflects common usage of the protocol referred to as "HTTP/1.0".

Table of Contents

1. Introduction	4
1.1 Purpose	4
1.2 Terminology	4
1.3 Overall Operation	6
1.4 HTTP and MIME	8
2. Notational Conventions and Generic Grammar	8
2.1 Augmented BNF	8
2.2 Basic Rules	10
3. Protocol Parameters	12

3.1	HTTP Version	12
3.2	Uniform Resource Identifiers	14
3.2.1	General Syntax	14
3.2.2	http URL	15
3.3	Date/Time Formats	15
3.4	Character Sets	17
3.5	Content Codings	18
3.6	Media Types	19
3.6.1	Canonicalization and Text Defaults	19
3.6.2	Multipart Types	20
3.7	Product Tokens	20
4.	HTTP Message	21
4.1	Message Types	21
4.2	Message Headers	22
4.3	General Header Fields	23
5.	Request	23
5.1	Request-Line	23
5.1.1	Method	24
5.1.2	Request-URI	24
5.2	Request Header Fields	25
6.	Response	25
6.1	Status-Line	26
6.1.1	Status Code and Reason Phrase	26
6.2	Response Header Fields	28
7.	Entity	28
7.1	Entity Header Fields	29
7.2	Entity Body	29
7.2.1	Type	29
7.2.2	Length	30
8.	Method Definitions	30
8.1	GET	31
8.2	HEAD	31
8.3	POST	31
9.	Status Code Definitions	32
9.1	Informational 1xx	32
9.2	Successful 2xx	32
9.3	Redirection 3xx	34
9.4	Client Error 4xx	35
9.5	Server Error 5xx	37
10.	Header Field Definitions	37
10.1	Allow	38
10.2	Authorization	38
10.3	Content-Encoding	39
10.4	Content-Length	39
10.5	Content-Type	40
10.6	Date	40
10.7	Expires	41
10.8	From	42

10.9	If-Modified-Since	42
10.10	Last-Modified	43
10.11	Location	44
10.12	Pragma	44
10.13	Referer	44
10.14	Server	45
10.15	User-Agent	46
10.16	WWW-Authenticate	46
11.	Access Authentication	47
11.1	Basic Authentication Scheme	48
12.	Security Considerations	49
12.1	Authentication of Clients	49
12.2	Safe Methods	49
12.3	Abuse of Server Log Information	50
12.4	Transfer of Sensitive Information	50
12.5	Attacks Based On File and Path Names	51
13.	Acknowledgments	51
14.	References	52
15.	Authors' Addresses	54
Appendix A.	Internet Media Type message/http	55
Appendix B.	Tolerant Applications	55
Appendix C.	Relationship to MIME	56
C.1	Conversion to Canonical Form	56
C.2	Conversion of Date Formats	57
C.3	Introduction of Content-Encoding	57
C.4	No Content-Transfer-Encoding	57
C.5	HTTP Header Fields in Multipart Body-Parts	57
Appendix D.	Additional Features	57
D.1	Additional Request Methods	58
D.1.1	PUT	58
D.1.2	DELETE	58
D.1.3	LINK	58
D.1.4	UNLINK	58
D.2	Additional Header Field Definitions	58
D.2.1	Accept	58
D.2.2	Accept-Charset	59
D.2.3	Accept-Encoding	59
D.2.4	Accept-Language	59
D.2.5	Content-Language	59
D.2.6	Link	59
D.2.7	MIME-Version	59
D.2.8	Retry-After	60
D.2.9	Title	60
D.2.10	URI	60

1. Introduction

1.1 Purpose

The Hypertext Transfer Protocol (HTTP) is an application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems. HTTP has been in use by the World-Wide Web global information initiative since 1990. This specification reflects common usage of the protocol referred to as "HTTP/1.0". This specification describes the features that seem to be consistently implemented in most HTTP/1.0 clients and servers. The specification is split into two sections. Those features of HTTP for which implementations are usually consistent are described in the main body of this document. Those features which have few or inconsistent implementations are listed in Appendix D.

Practical information systems require more functionality than simple retrieval, including search, front-end update, and annotation. HTTP allows an open-ended set of methods to be used to indicate the purpose of a request. It builds on the discipline of reference provided by the Uniform Resource Identifier (URI) [2], as a location (URL) [4] or name (URN) [16], for indicating the resource on which a method is to be applied. Messages are passed in a format similar to that used by Internet Mail [7] and the Multipurpose Internet Mail Extensions (MIME) [5].

HTTP is also used as a generic protocol for communication between user agents and proxies/gateways to other Internet protocols, such as SMTP [12], NNTP [11], FTP [14], Gopher [1], and WAIS [8], allowing basic hypermedia access to resources available from diverse applications and simplifying the implementation of user agents.

1.2 Terminology

This specification uses a number of terms to refer to the roles played by participants in, and objects of, the HTTP communication.

connection

A transport layer virtual circuit established between two application programs for the purpose of communication.

message

The basic unit of HTTP communication, consisting of a structured sequence of octets matching the syntax defined in Section 4 and transmitted via the connection.

request

An HTTP request message (as defined in Section 5).

response

An HTTP response message (as defined in Section 6).

resource

A network data object or service which can be identified by a URI (Section 3.2).

entity

A particular representation or rendition of a data resource, or reply from a service resource, that may be enclosed within a request or response message. An entity consists of metainformation in the form of entity headers and content in the form of an entity body.

client

An application program that establishes connections for the purpose of sending requests.

user agent

The client which initiates a request. These are often browsers, editors, spiders (web-traversing robots), or other end user tools.

server

An application program that accepts connections in order to service requests by sending back responses.

origin server

The server on which a given resource resides or is to be created.

proxy

An intermediary program which acts as both a server and a client for the purpose of making requests on behalf of other clients. Requests are serviced internally or by passing them, with possible translation, on to other servers. A proxy must interpret and, if necessary, rewrite a request message before

forwarding it. Proxies are often used as client-side portals through network firewalls and as helper applications for handling requests via protocols not implemented by the user agent.

gateway

A server which acts as an intermediary for some other server. Unlike a proxy, a gateway receives requests as if it were the origin server for the requested resource; the requesting client may not be aware that it is communicating with a gateway. Gateways are often used as server-side portals through network firewalls and as protocol translators for access to resources stored on non-HTTP systems.

tunnel

A tunnel is an intermediary program which is acting as a blind relay between two connections. Once active, a tunnel is not considered a party to the HTTP communication, though the tunnel may have been initiated by an HTTP request. The tunnel ceases to exist when both ends of the relayed connections are closed. Tunnels are used when a portal is necessary and the intermediary cannot, or should not, interpret the relayed communication.

cache

A program's local store of response messages and the subsystem that controls its message storage, retrieval, and deletion. A cache stores cachable responses in order to reduce the response time and network bandwidth consumption on future, equivalent requests. Any client or server may include a cache, though a cache cannot be used by a server while it is acting as a tunnel.

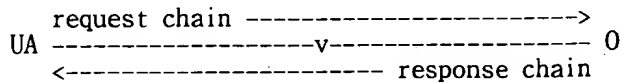
Any given program may be capable of being both a client and a server; our use of these terms refers only to the role being performed by the program for a particular connection, rather than to the program's capabilities in general. Likewise, any server may act as an origin server, proxy, gateway, or tunnel, switching behavior based on the nature of each request.

1.3 Overall Operation

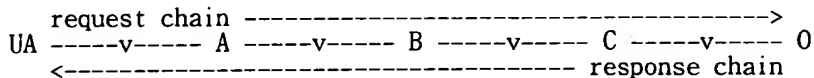
The HTTP protocol is based on a request/response paradigm. A client establishes a connection with a server and sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content. The server responds with a

status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity metainformation, and possible body content.

Most HTTP communication is initiated by a user agent and consists of a request to be applied to a resource on some origin server. In the simplest case, this may be accomplished via a single connection (v) between the user agent (UA) and the origin server (O).



A more complicated situation occurs when one or more intermediaries are present in the request/response chain. There are three common forms of intermediary: proxy, gateway, and tunnel. A proxy is a forwarding agent, receiving requests for a URI in its absolute form, rewriting all or parts of the message, and forwarding the reformatted request toward the server identified by the URI. A gateway is a receiving agent, acting as a layer above some other server(s) and, if necessary, translating the requests to the underlying server's protocol. A tunnel acts as a relay point between two connections without changing the messages; tunnels are used when the communication needs to pass through an intermediary (such as a firewall) even when the intermediary cannot understand the contents of the messages.



The figure above shows three intermediaries (A, B, and C) between the user agent and origin server. A request or response message that travels the whole chain must pass through four separate connections. This distinction is important because some HTTP communication options may apply only to the connection with the nearest, non-tunnel neighbor, only to the end-points of the chain, or to all connections along the chain. Although the diagram is linear, each participant may be engaged in multiple, simultaneous communications. For example, B may be receiving requests from many clients other than A, and/or forwarding requests to servers other than C, at the same time that it is handling A's request.

Any party to the communication which is not acting as a tunnel may employ an internal cache for handling requests. The effect of a cache is that the request/response chain is shortened if one of the participants along the chain has a cached response applicable to that request. The following illustrates the resulting chain if B has a

cached copy of an earlier response from O (via C) for a request which has not been cached by UA or A.

```

      request chain ----->
UA  -----v----- A -----v----- B - - - - - C - - - - - O
<----- response chain

```

Not all responses are cachable, and some requests may contain modifiers which place special requirements on cache behavior. Some HTTP/1.0 applications use heuristics to describe what is or is not a "cachable" response, but these rules are not standardized.

On the Internet, HTTP communication generally takes place over TCP/IP connections. The default port is TCP 80 [15], but other ports can be used. This does not preclude HTTP from being implemented on top of any other protocol on the Internet, or on other networks. HTTP only presumes a reliable transport; any protocol that provides such guarantees can be used, and the mapping of the HTTP/1.0 request and response structures onto the transport data units of the protocol in question is outside the scope of this specification.

Except for experimental applications, current practice requires that the connection be established by the client prior to each request and closed by the server after sending the response. Both clients and servers should be aware that either party may close the connection prematurely, due to user action, automated time-out, or program failure, and should handle such closing in a predictable fashion. In any case, the closing of the connection by either or both parties always terminates the current request, regardless of its status.

1.4 HTTP and MIME

HTTP/1.0 uses many of the constructs defined for MIME, as defined in RFC 1521 [5]. Appendix C describes the ways in which the context of HTTP allows for different use of Internet Media Types than is typically found in Internet mail, and gives the rationale for those differences.

2. Notational Conventions and Generic Grammar

2.1 Augmented BNF

All of the mechanisms specified in this document are described in both prose and an augmented Backus-Naur Form (BNF) similar to that used by RFC 822 [7]. Implementors will need to be familiar with the notation in order to understand this specification. The augmented BNF includes the following constructs:

name = definition

The name of a rule is simply the name itself (without any enclosing "<" and ">") and is separated from its definition by the equal character "=". Whitespace is only significant in that indentation of continuation lines is used to indicate a rule definition that spans more than one line. Certain basic rules are in uppercase, such as SP, LWS, HT, CRLF, DIGIT, ALPHA, etc. Angle brackets are used within definitions whenever their presence will facilitate discerning the use of rule names.

"literal"

Quotation marks surround literal text. Unless stated otherwise, the text is case-insensitive.

rule1 | rule2

Elements separated by a bar ("|") are alternatives, e.g., "yes | no" will accept yes or no.

(rule1 rule2)

Elements enclosed in parentheses are treated as a single element. Thus, "(elem (foo | bar) elem)" allows the token sequences "elem foo elem" and "elem bar elem".

*rule

The character "*" preceding an element indicates repetition. The full form is "<n>*<m>element" indicating at least <n> and at most <m> occurrences of element. Default values are 0 and infinity so that "(element)" allows any number, including zero; "1*element" requires at least one; and "1*2element" allows one or two.

[rule]

Square brackets enclose optional elements; "[foo bar]" is equivalent to "1(foo bar)".

N rule

Specific repetition: "<n>(element)" is equivalent to "<n>*<n>(element)"; that is, exactly <n> occurrences of (element). Thus 2DIGIT is a 2-digit number, and 3ALPHA is a string of three alphabetic characters.

#rule

A construct "#" is defined, similar to "*", for defining lists of elements. The full form is "<n>#<m>element" indicating at least <n> and at most <m> elements, each separated by one or more commas (",") and optional linear whitespace (LWS). This makes the usual form of lists very easy; a rule such as "(*LWS element *(*LWS "," *LWS element))" can be shown as "1#element". Wherever this construct is used, null elements are allowed, but do not contribute to the count of elements present. That is, "(element), , (element)" is permitted, but counts as only two elements. Therefore, where at least one element is required, at least one non-null element must be present. Default values are 0 and infinity so that "#(element)" allows any number, including zero; "1#element" requires at least one; and "1#2element" allows one or two.

; comment

A semi-colon, set off some distance to the right of rule text, starts a comment that continues to the end of line. This is a simple way of including useful notes in parallel with the specifications.

implied *LWS

The grammar described by this specification is word-based. Except where noted otherwise, linear whitespace (LWS) can be included between any two adjacent words (token or quoted-string), and between adjacent tokens and delimiters (tspecials), without changing the interpretation of a field. At least one delimiter (tspecials) must exist between any two tokens, since they would otherwise be interpreted as a single token. However, applications should attempt to follow "common form" when generating HTTP constructs, since there exist some implementations that fail to accept anything beyond the common forms.

2.2 Basic Rules

The following rules are used throughout this specification to describe basic parsing constructs. The US-ASCII coded character set is defined by [17].

OCTET	= <any 8-bit sequence of data>
CHAR	= <any US-ASCII character (octets 0 - 127)>
UPALPHA	= <any US-ASCII uppercase letter "A".."Z">
LOALPHA	= <any US-ASCII lowercase letter "a".."z">

ALPHA	= UPALPHA LOALPHA
DIGIT	= <any US-ASCII digit "0".."9">
CTL	= <any US-ASCII control character (octets 0 - 31) and DEL (127)>
CR	= <US-ASCII CR, carriage return (13)>
LF	= <US-ASCII LF, linefeed (10)>
SP	= <US-ASCII SP, space (32)>
HT	= <US-ASCII HT, horizontal-tab (9)>
<">	= <US-ASCII double-quote mark (34)>

HTTP/1.0 defines the octet sequence CR LF as the end-of-line marker for all protocol elements except the Entity-Body (see Appendix B for tolerant applications). The end-of-line marker within an Entity-Body is defined by its associated media type, as described in Section 3.6.

CRLF = CR LF

HTTP/1.0 headers may be folded onto multiple lines if each continuation line begins with a space or horizontal tab. All linear whitespace, including folding, has the same semantics as SP.

LWS = [CRLF] 1*(SP | HT)

However, folding of header lines is not expected by some applications, and should not be generated by HTTP/1.0 applications.

The TEXT rule is only used for descriptive field contents and values that are not intended to be interpreted by the message parser. Words of *TEXT may contain octets from character sets other than US-ASCII.

TEXT = <any OCTET except CTLs,
but including LWS>

Recipients of header field TEXT containing octets outside the US-ASCII character set may assume that they represent ISO-8859-1 characters.

Hexadecimal numeric characters are used in several protocol elements.

HEX = "A" | "B" | "C" | "D" | "E" | "F"
| "a" | "b" | "c" | "d" | "e" | "f" | DIGIT

Many HTTP/1.0 header field values consist of words separated by LWS or special characters. These special characters must be in a quoted string to be used within a parameter value.

word = token | quoted-string

```

token      = 1*<any CHAR except CTLs or tspecials>

tspecials  = "(" | ")" | "<" | ">" | "@"
             | "\"" | "." | ":" | ";" | "\\" | "<" | ">"
             | "/" | "[" | "]" | "?" | "="
             | "!" | "}" | " " | SP | HT

```

Comments may be included in some HTTP header fields by surrounding the comment text with parentheses. Comments are only allowed in fields containing "comment" as part of their field value definition. In all other fields, parentheses are considered part of the field value.

```

comment    = "(" *( ctext | comment ) ")"
ctext      = <any TEXT excluding "(" and ")">

```

A string of text is parsed as a single word if it is quoted using double-quote marks.

```

quoted-string = ( "<" *(qdtex) ">" )

qdtex        = <any CHAR except "<" and CTLs,
               but including LWS>

```

Single-character quoting using the backslash ("\") character is not permitted in HTTP/1.0.

3. Protocol Parameters

3.1 HTTP Version

HTTP uses a "<major>.<minor>" numbering scheme to indicate versions of the protocol. The protocol versioning policy is intended to allow the sender to indicate the format of a message and its capacity for understanding further HTTP communication, rather than the features obtained via that communication. No change is made to the version number for the addition of message components which do not affect communication behavior or which only add to extensible field values. The <minor> number is incremented when the changes made to the protocol add features which do not change the general message parsing algorithm, but which may add to the message semantics and imply additional capabilities of the sender. The <major> number is incremented when the format of a message within the protocol is changed.

The version of an HTTP message is indicated by an HTTP-Version field in the first line of the message. If the protocol version is not specified, the recipient must assume that the message is in the

simple HTTP/0.9 format.

HTTP-Version = "HTTP" "/" 1*DIGIT "." 1*DIGIT

Note that the major and minor numbers should be treated as separate integers and that each may be incremented higher than a single digit. Thus, HTTP/2.4 is a lower version than HTTP/2.13, which in turn is lower than HTTP/12.3. Leading zeros should be ignored by recipients and never generated by senders.

This document defines both the 0.9 and 1.0 versions of the HTTP protocol. Applications sending Full-Request or Full-Response messages, as defined by this specification, must include an HTTP-Version of "HTTP/1.0".

HTTP/1.0 servers must:

- o recognize the format of the Request-Line for HTTP/0.9 and HTTP/1.0 requests;
- o understand any valid request in the format of HTTP/0.9 or HTTP/1.0;
- o respond appropriately with a message in the same protocol version used by the client.

HTTP/1.0 clients must:

- o recognize the format of the Status-Line for HTTP/1.0 responses;
- o understand any valid response in the format of HTTP/0.9 or HTTP/1.0.

Proxy and gateway applications must be careful in forwarding requests that are received in a format different than that of the application's native HTTP version. Since the protocol version indicates the protocol capability of the sender, a proxy/gateway must never send a message with a version indicator which is greater than its native version; if a higher version request is received, the proxy/gateway must either downgrade the request version or respond with an error. Requests with a version lower than that of the application's native format may be upgraded before being forwarded; the proxy/gateway's response to that request must follow the server requirements listed above.

3.2 Uniform Resource Identifiers

URIs have been known by many names: WWW addresses, Universal Document Identifiers, Universal Resource Identifiers [2], and finally the combination of Uniform Resource Locators (URL) [4] and Names (URN) [16]. As far as HTTP is concerned, Uniform Resource Identifiers are simply formatted strings which identify--via name, location, or any other characteristic--a network resource.

3.2.1 General Syntax

URIs in HTTP can be represented in absolute form or relative to some known base URI [9], depending upon the context of their use. The two forms are differentiated by the fact that absolute URIs always begin with a scheme name followed by a colon.

```

URI           = ( absoluteURI | relativeURI ) [ "#" fragment ]
absoluteURI   = scheme ":" *( uchar | reserved )
relativeURI   = net_path | abs_path | rel_path

net_path      = "//" net_loc [ abs_path ]
abs_path      = "/" rel_path
rel_path      = [ path ] [ ";" params ] [ "?" query ]

path          = fsegment *( "/" segment )
fsegment      = 1*pchar
segment       = *pchar

params        = param *( ";" param )
param         = *( pchar | "/" )

scheme        = 1*( ALPHA | DIGIT | "+" | "-" | "." )
net_loc       = *( pchar | ";" | "?" )
query         = *( uchar | reserved )
fragment      = *( uchar | reserved )

pchar         = uchar | ":" | "@" | "&" | "=" | "+"
uchar         = unreserved | escape
unreserved    = ALPHA | DIGIT | safe | extra | national

escape        = "%" HEX HEX
reserved      = "." | "/" | "?" | ":" | "@" | "&" | "=" | "+"
extra         = "!" | "*" | "'" | "(" | ")" | ","
safe          = "$" | "-" | "_" | "."
unsafe        = CTL | SP | "<" | ">" | "#" | "%" | "<" | ">"
national      = <any OCTET excluding ALPHA, DIGIT,
```

reserved, extra, safe, and unsafe>

For definitive information on URL syntax and semantics, see RFC 1738 [4] and RFC 1808 [9]. The BNF above includes national characters not allowed in valid URLs as specified by RFC 1738, since HTTP servers are not restricted in the set of unreserved characters allowed to represent the `rel_path` part of addresses, and HTTP proxies may receive requests for URIs not defined by RFC 1738.

3.2.2 http URL

The "http" scheme is used to locate network resources via the HTTP protocol. This section defines the scheme-specific syntax and semantics for http URLs.

```

http_URL      = "http:" "/" host [ ":" port ] [ abs_path ]
host          = <A legal Internet host domain name
               or IP address (in dotted-decimal form),
               as defined by Section 2.1 of RFC 1123>
port          = *DIGIT

```

If the port is empty or not given, port 80 is assumed. The semantics are that the identified resource is located at the server listening for TCP connections on that port of that host, and the Request-URI for the resource is `abs_path`. If the `abs_path` is not present in the URL, it must be given as "/" when used as a Request-URI (Section 5.1.2).

Note: Although the HTTP protocol is independent of the transport layer protocol, the http URL only identifies resources by their TCP location, and thus non-TCP resources must be identified by some other URI scheme.

The canonical form for "http" URLs is obtained by converting any UPALPHA characters in host to their LOALPHA equivalent (hostnames are case-insensitive), eliding the [":" port] if the port is 80, and replacing an empty `abs_path` with "/".

3.3 Date/Time Formats

HTTP/1.0 applications have historically allowed three different formats for the representation of date/time stamps:

```

Sun, 06 Nov 1994 08:49:37 GMT    ; RFC 822, updated by RFC 1123
Sunday, 06-Nov-94 08:49:37 GMT   ; RFC 850, obsoleted by RFC 1036
Sun Nov  6 08:49:37 1994         ; ANSI C's asctime() format

```

The first format is preferred as an Internet standard and represents a fixed-length subset of that defined by RFC 1123 [6] (an update to RFC 822 [7]). The second format is in common use, but is based on the obsolete RFC 850 [10] date format and lacks a four-digit year. HTTP/1.0 clients and servers that parse the date value should accept all three formats, though they must never generate the third (asctime) format.

Note: Recipients of date values are encouraged to be robust in accepting date values that may have been generated by non-HTTP applications, as is sometimes the case when retrieving or posting messages via proxies/gateways to SMTP or NNTP.

All HTTP/1.0 date/time stamps must be represented in Universal Time (UT), also known as Greenwich Mean Time (GMT), without exception. This is indicated in the first two formats by the inclusion of "GMT" as the three-letter abbreviation for time zone, and should be assumed when reading the asctime format.

HTTP-date = rfc1123-date | rfc850-date | asctime-date

rfc1123-date = wkday ", " SP date1 SP time SP "GMT"

rfc850-date = weekday ", " SP date2 SP time SP "GMT"

asctime-date = wkday SP date3 SP time SP 4DIGIT

date1 = 2DIGIT SP month SP 4DIGIT
; day month year (e.g., 02 Jun 1982)

date2 = 2DIGIT "-" month "-" 2DIGIT
; day-month-year (e.g., 02-Jun-82)

date3 = month SP (2DIGIT | (SP 1DIGIT))
; month day (e.g., Jun 2)

time = 2DIGIT ":" 2DIGIT ":" 2DIGIT
; 00:00:00 - 23:59:59

wkday = "Mon" | "Tue" | "Wed"
| "Thu" | "Fri" | "Sat" | "Sun"

weekday = "Monday" | "Tuesday" | "Wednesday"
| "Thursday" | "Friday" | "Saturday" | "Sunday"

month = "Jan" | "Feb" | "Mar" | "Apr"
| "May" | "Jun" | "Jul" | "Aug"
| "Sep" | "Oct" | "Nov" | "Dec"

Note: HTTP requirements for the date/time stamp format apply only to their usage within the protocol stream. Clients and servers are not required to use these formats for user

presentation, request logging, etc.

3.4 Character Sets

HTTP uses the same definition of the term "character set" as that described for MIME:

The term "character set" is used in this document to refer to a method used with one or more tables to convert a sequence of octets into a sequence of characters. Note that unconditional conversion in the other direction is not required, in that not all characters may be available in a given character set and a character set may provide more than one sequence of octets to represent a particular character. This definition is intended to allow various kinds of character encodings, from simple single-table mappings such as US-ASCII to complex table switching methods such as those that use ISO 2022's techniques. However, the definition associated with a MIME character set name must fully specify the mapping to be performed from octets to characters. In particular, use of external profiling information to determine the exact mapping is not permitted.

Note: This use of the term "character set" is more commonly referred to as a "character encoding." However, since HTTP and MIME share the same registry, it is important that the terminology also be shared.

HTTP character sets are identified by case-insensitive tokens. The complete set of tokens are defined by the IANA Character Set registry [15]. However, because that registry does not define a single, consistent token for each character set, we define here the preferred names for those character sets most likely to be used with HTTP entities. These character sets include those registered by RFC 1521 [5] -- the US-ASCII [17] and ISO-8859 [18] character sets -- and other names specifically recommended for use within MIME charset parameters.

```
charset = "US-ASCII"  
         | "ISO-8859-1" | "ISO-8859-2" | "ISO-8859-3"  
         | "ISO-8859-4" | "ISO-8859-5" | "ISO-8859-6"  
         | "ISO-8859-7" | "ISO-8859-8" | "ISO-8859-9"  
         | "ISO-2022-JP" | "ISO-2022-JP-2" | "ISO-2022-KR"  
         | "UNICODE-1-1" | "UNICODE-1-1-UTF-7" | "UNICODE-1-1-UTF-8"  
         token
```

Although HTTP allows an arbitrary token to be used as a charset value, any token that has a predefined value within the IANA Character Set registry [15] must represent the character set defined

by that registry. Applications should limit their use of character sets to those defined by the IANA registry.

The character set of an entity body should be labelled as the lowest common denominator of the character codes used within that body, with the exception that no label is preferred over the labels US-ASCII or ISO-8859-1.

3.5 Content Codings

Content coding values are used to indicate an encoding transformation that has been applied to a resource. Content codings are primarily used to allow a document to be compressed or encrypted without losing the identity of its underlying media type. Typically, the resource is stored in this encoding and only decoded before rendering or analogous usage.

content-coding = "x-gzip" | "x-compress" | token

Note: For future compatibility, HTTP/1.0 applications should consider "gzip" and "compress" to be equivalent to "x-gzip" and "x-compress", respectively.

All content-coding values are case-insensitive. HTTP/1.0 uses content-coding values in the Content-Encoding (Section 10.3) header field. Although the value describes the content-coding, what is more important is that it indicates what decoding mechanism will be required to remove the encoding. Note that a single program may be capable of decoding multiple content-coding formats. Two values are defined by this specification:

x-gzip

An encoding format produced by the file compression program "gzip" (GNU zip) developed by Jean-loup Gailly. This format is typically a Lempel-Ziv coding (LZ77) with a 32 bit CRC.

x-compress

The encoding format produced by the file compression program "compress". This format is an adaptive Lempel-Ziv-Welch coding (LZW).

Note: Use of program names for the identification of encoding formats is not desirable and should be discouraged for future encodings. Their use here is representative of historical practice, not good design.

3.6 Media Types

HTTP uses Internet Media Types [13] in the Content-Type header field (Section 10.5) in order to provide open and extensible data typing.

```
media-type    = type "/" subtype *( ";" parameter )
type          = token
subtype       = token
```

Parameters may follow the type/subtype in the form of attribute/value pairs.

```
parameter     = attribute "=" value
attribute      = token
value         = token | quoted-string
```

The type, subtype, and parameter attribute names are case-insensitive. Parameter values may or may not be case-sensitive, depending on the semantics of the parameter name. LWS must not be generated between the type and subtype, nor between an attribute and its value. Upon receipt of a media type with an unrecognized parameter, a user agent should treat the media type as if the unrecognized parameter and its value were not present.

Some older HTTP applications do not recognize media type parameters. HTTP/1.0 applications should only use media type parameters when they are necessary to define the content of a message.

Media-type values are registered with the Internet Assigned Number Authority (IANA [15]). The media type registration process is outlined in RFC 1590 [13]. Use of non-registered media types is discouraged.

3.6.1 Canonicalization and Text Defaults

Internet media types are registered with a canonical form. In general, an Entity-Body transferred via HTTP must be represented in the appropriate canonical form prior to its transmission. If the body has been encoded with a Content-Encoding, the underlying data should be in canonical form prior to being encoded.

Media subtypes of the "text" type use CRLF as the text line break when in canonical form. However, HTTP allows the transport of text media with plain CR or LF alone representing a line break when used consistently within the Entity-Body. HTTP applications must accept CRLF, bare CR, and bare LF as being representative of a line break in text media received via HTTP.

In addition, if the text media is represented in a character set that does not use octets 13 and 10 for CR and LF respectively, as is the case for some multi-byte character sets, HTTP allows the use of whatever octet sequences are defined by that character set to represent the equivalent of CR and LF for line breaks. This flexibility regarding line breaks applies only to text media in the Entity-Body; a bare CR or LF should not be substituted for CRLF within any of the HTTP control structures (such as header fields and multipart boundaries).

The "charset" parameter is used with some media types to define the character set (Section 3.4) of the data. When no explicit charset parameter is provided by the sender, media subtypes of the "text" type are defined to have a default charset value of "ISO-8859-1" when received via HTTP. Data in character sets other than "ISO-8859-1" or its subsets must be labelled with an appropriate charset value in order to be consistently interpreted by the recipient.

Note: Many current HTTP servers provide data using charsets other than "ISO-8859-1" without proper labelling. This situation reduces interoperability and is not recommended. To compensate for this, some HTTP user agents provide a configuration option to allow the user to change the default interpretation of the media type character set when no charset parameter is given.

3.6.2 Multipart Types

MIME provides for a number of "multipart" types -- encapsulations of several entities within a single message's Entity-Body. The multipart types registered by IANA [15] do not have any special meaning for HTTP/1.0, though user agents may need to understand each type in order to correctly interpret the purpose of each body-part. An HTTP user agent should follow the same or similar behavior as a MIME user agent does upon receipt of a multipart type. HTTP servers should not assume that all HTTP clients are prepared to handle multipart types.

All multipart types share a common syntax and must include a boundary parameter as part of the media type value. The message body is itself a protocol element and must therefore use only CRLF to represent line breaks between body-parts. Multipart body-parts may contain HTTP header fields which are significant to the meaning of that part.

3.7 Product Tokens

Product tokens are used to allow communicating applications to identify themselves via a simple product token, with an optional slash and version designator. Most fields using product tokens also allow subproducts which form a significant part of the application to

be listed, separated by whitespace. By convention, the products are listed in order of their significance for identifying the application.

```
product      = token ["/" product-version]
product-version = token
```

Examples:

User-Agent: CERN-LineMode/2.15 libwww/2.17b3

Server: Apache/0.8.4

Product tokens should be short and to the point -- use of them for advertizing or other non-essential information is explicitly forbidden. Although any token character may appear in a product-version, this token should only be used for a version identifier (i.e., successive versions of the same product should only differ in the product-version portion of the product value).

4. HTTP Message

4.1 Message Types

HTTP messages consist of requests from client to server and responses from server to client.

```
HTTP-message = Simple-Request      ; HTTP/0.9 messages
              | Simple-Response
              | Full-Request        ; HTTP/1.0 messages
              | Full-Response
```

Full-Request and Full-Response use the generic message format of RFC 822 [7] for transferring entities. Both messages may include optional header fields (also known as "headers") and an entity body. The entity body is separated from the headers by a null line (i.e., a line with nothing preceding the CRLF).

```
Full-Request = Request-Line        ; Section 5.1
              *( General-Header     ; Section 4.3
                | Request-Header    ; Section 5.2
                | Entity-Header )   ; Section 7.1
              CRLF
              [ Entity-Body ]       ; Section 7.2

Full-Response = Status-Line        ; Section 6.1
              *( General-Header     ; Section 4.3
                | Response-Header   ; Section 6.2
```

```

    | Entity-Header )      ; Section 7.1
    CRLF
    [ Entity-Body ]      ; Section 7.2

```

Simple-Request and Simple-Response do not allow the use of any header information and are limited to a single request method (GET).

Simple-Request = "GET" SP Request-URI CRLF

Simple-Response = [Entity-Body]

Use of the Simple-Request format is discouraged because it prevents the server from identifying the media type of the returned entity.

4.2 Message Headers

HTTP header fields, which include General-Header (Section 4.3), Request-Header (Section 5.2), Response-Header (Section 6.2), and Entity-Header (Section 7.1) fields, follow the same generic format as that given in Section 3.1 of RFC 822 [7]. Each header field consists of a name followed immediately by a colon (":"), a single space (SP) character, and the field value. Field names are case-insensitive. Header fields can be extended over multiple lines by preceding each extra line with at least one SP or HT, though this is not recommended.

HTTP-header = field-name ":" [field-value] CRLF

field-name = token

field-value = *(field-content | LWS)

field-content = <the OCTETs making up the field-value
and consisting of either *TEXT or combinations
of token, tspecials, and quoted-string>

The order in which header fields are received is not significant. However, it is "good practice" to send General-Header fields first, followed by Request-Header or Response-Header fields prior to the Entity-Header fields.

Multiple HTTP-header fields with the same field-name may be present in a message if and only if the entire field-value for that header field is defined as a comma-separated list [i.e., #(values)]. It must be possible to combine the multiple header fields into one "field-name: field-value" pair, without changing the semantics of the message, by appending each subsequent field-value to the first, each separated by a comma.

4.3 General Header Fields

There are a few header fields which have general applicability for both request and response messages, but which do not apply to the entity being transferred. These headers apply only to the message being transmitted.

```
General-Header = Date           ; Section 10.6
                | Pragma        ; Section 10.12
```

General header field names can be extended reliably only in combination with a change in the protocol version. However, new or experimental header fields may be given the semantics of general header fields if all parties in the communication recognize them to be general header fields. Unrecognized header fields are treated as Entity-Header fields.

5. Request

A request message from a client to a server includes, within the first line of that message, the method to be applied to the resource, the identifier of the resource, and the protocol version in use. For backwards compatibility with the more limited HTTP/0.9 protocol, there are two valid formats for an HTTP request:

```
Request        = Simple-Request | Full-Request

Simple-Request = "GET" SP Request-URI CRLF

Full-Request   = Request-Line           ; Section 5.1
                *( General-Header       ; Section 4.3
                  | Request-Header      ; Section 5.2
                  | Entity-Header )     ; Section 7.1
                CRLF
                [ Entity-Body ]         ; Section 7.2
```

If an HTTP/1.0 server receives a Simple-Request, it must respond with an HTTP/0.9 Simple-Response. An HTTP/1.0 client capable of receiving a Full-Response should never generate a Simple-Request.

5.1 Request-Line

The Request-Line begins with a method token, followed by the Request-URI and the protocol version, and ending with CRLF. The elements are separated by SP characters. No CR or LF are allowed except in the final CRLF sequence.

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

Note that the difference between a Simple-Request and the Request-Line of a Full-Request is the presence of the HTTP-Version field and the availability of methods other than GET.

5.1.1 Method

The Method token indicates the method to be performed on the resource identified by the Request-URI. The method is case-sensitive.

```
Method      = "GET"                ; Section 8.1
             | "HEAD"              ; Section 8.2
             | "POST"              ; Section 8.3
             | extension-method
```

extension-method = token

The list of methods acceptable by a specific resource can change dynamically; the client is notified through the return code of the response if a method is not allowed on a resource. Servers should return the status code 501 (not implemented) if the method is unrecognized or not implemented.

The methods commonly used by HTTP/1.0 applications are fully defined in Section 8.

5.1.2 Request-URI

The Request-URI is a Uniform Resource Identifier (Section 3.2) and identifies the resource upon which to apply the request.

```
Request-URI  = absoluteURI | abs_path
```

The two options for Request-URI are dependent on the nature of the request.

The absoluteURI form is only allowed when the request is being made to a proxy. The proxy is requested to forward the request and return the response. If the request is GET or HEAD and a prior response is cached, the proxy may use the cached message if it passes any restrictions in the Expires header field. Note that the proxy may forward the request on to another proxy or directly to the server specified by the absoluteURI. In order to avoid request loops, a proxy must be able to recognize all of its server names, including any aliases, local variations, and the numeric IP address. An example Request-Line would be:

```
GET http://www.w3.org/pub/WWW/TheProject.html HTTP/1.0
```

The most common form of Request-URI is that used to identify a resource on an origin server or gateway. In this case, only the absolute path of the URI is transmitted (see Section 3.2.1, `abs_path`). For example, a client wishing to retrieve the resource above directly from the origin server would create a TCP connection to port 80 of the host "www.w3.org" and send the line:

```
GET /pub/WWW/TheProject.html HTTP/1.0
```

followed by the remainder of the Full-Request. Note that the absolute path cannot be empty; if none is present in the original URI, it must be given as "/" (the server root).

The Request-URI is transmitted as an encoded string, where some characters may be escaped using the "% HEX HEX" encoding defined by RFC 1738 [4]. The origin server must decode the Request-URI in order to properly interpret the request.

5.2 Request Header Fields

The request header fields allow the client to pass additional information about the request, and about the client itself, to the server. These fields act as request modifiers, with semantics equivalent to the parameters on a programming language method (procedure) invocation.

Request-Header	=	Authorization	;	Section 10.2
		From	;	Section 10.8
		If-Modified-Since	;	Section 10.9
		Referer	;	Section 10.13
		User-Agent	;	Section 10.15

Request-Header field names can be extended reliably only in combination with a change in the protocol version. However, new or experimental header fields may be given the semantics of request header fields if all parties in the communication recognize them to be request header fields. Unrecognized header fields are treated as Entity-Header fields.

6. Response

After receiving and interpreting a request message, a server responds in the form of an HTTP response message.

Response = Simple-Response | Full-Response

Simple-Response = [Entity-Body]

```

Full-Response = Status-Line      ; Section 6.1
                *( General-Header ; Section 4.3
                  | Response-Header ; Section 6.2
                  | Entity-Header ) ; Section 7.1
                CRLF
                [ Entity-Body ]   ; Section 7.2

```

A Simple-Response should only be sent in response to an HTTP/0.9 Simple-Request or if the server only supports the more limited HTTP/0.9 protocol. If a client sends an HTTP/1.0 Full-Request and receives a response that does not begin with a Status-Line, it should assume that the response is a Simple-Response and parse it accordingly. Note that the Simple-Response consists only of the entity body and is terminated by the server closing the connection.

6.1 Status-Line

The first line of a Full-Response message is the Status-Line, consisting of the protocol version followed by a numeric status code and its associated textual phrase, with each element separated by SP characters. No CR or LF is allowed except in the final CRLF sequence.

Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF

Since a status line always begins with the protocol version and status code

"HTTP/" 1*DIGIT "." 1*DIGIT SP 3DIGIT SP

(e.g., "HTTP/1.0 200 "), the presence of that expression is sufficient to differentiate a Full-Response from a Simple-Response. Although the Simple-Response format may allow such an expression to occur at the beginning of an entity body, and thus cause a misinterpretation of the message if it was given in response to a Full-Request, most HTTP/0.9 servers are limited to responses of type "text/html" and therefore would never generate such a response.

6.1.1 Status Code and Reason Phrase

The Status-Code element is a 3-digit integer result code of the attempt to understand and satisfy the request. The Reason-Phrase is intended to give a short textual description of the Status-Code. The Status-Code is intended for use by automata and the Reason-Phrase is intended for the human user. The client is not required to examine or display the Reason-Phrase.

The first digit of the Status-Code defines the class of response. The last two digits do not have any categorization role. There are 5 values for the first digit:

- o 1xx: Informational - Not used, but reserved for future use
- o 2xx: Success - The action was successfully received, understood, and accepted.
- o 3xx: Redirection - Further action must be taken in order to complete the request
- o 4xx: Client Error - The request contains bad syntax or cannot be fulfilled
- o 5xx: Server Error - The server failed to fulfill an apparently valid request

The individual values of the numeric status codes defined for HTTP/1.0, and an example set of corresponding Reason-Phrase's, are presented below. The reason phrases listed here are only recommended -- they may be replaced by local equivalents without affecting the protocol. These codes are fully defined in Section 9.

Status-Code	=	"200"	:	OK
		"201"	:	Created
		"202"	:	Accepted
		"204"	:	No Content
		"301"	:	Moved Permanently
		"302"	:	Moved Temporarily
		"304"	:	Not Modified
		"400"	:	Bad Request
		"401"	:	Unauthorized
		"403"	:	Forbidden
		"404"	:	Not Found
		"500"	:	Internal Server Error
		"501"	:	Not Implemented
		"502"	:	Bad Gateway
		"503"	:	Service Unavailable
		extension-code		

extension-code = 3DIGIT

Reason-Phrase = *<TEXT, excluding CR, LF>

HTTP status codes are extensible, but the above codes are the only ones generally recognized in current practice. HTTP applications are not required to understand the meaning of all registered status

codes, though such understanding is obviously desirable. However, applications must understand the class of any status code, as indicated by the first digit, and treat any unrecognized response as being equivalent to the x00 status code of that class, with the exception that an unrecognized response must not be cached. For example, if an unrecognized status code of 431 is received by the client, it can safely assume that there was something wrong with its request and treat the response as if it had received a 400 status code. In such cases, user agents should present to the user the entity returned with the response, since that entity is likely to include human-readable information which will explain the unusual status.

6.2 Response Header Fields

The response header fields allow the server to pass additional information about the response which cannot be placed in the Status-Line. These header fields give information about the server and about further access to the resource identified by the Request-URI.

Response-Header	= Location	; Section 10.11
	Server	; Section 10.14
	WWW-Authenticate	; Section 10.16

Response-Header field names can be extended reliably only in combination with a change in the protocol version. However, new or experimental header fields may be given the semantics of response header fields if all parties in the communication recognize them to be response header fields. Unrecognized header fields are treated as Entity-Header fields.

7. Entity

Full-Request and Full-Response messages may transfer an entity within some requests and responses. An entity consists of Entity-Header fields and (usually) an Entity-Body. In this section, both sender and recipient refer to either the client or the server, depending on who sends and who receives the entity.

7.1 Entity Header Fields

Entity-Header fields define optional metainformation about the Entity-Body or, if no body is present, about the resource identified by the request.

Entity-Header	= Allow	; Section 10.1
	Content-Encoding	; Section 10.3
	Content-Length	; Section 10.4
	Content-Type	; Section 10.5
	Expires	; Section 10.7
	Last-Modified	; Section 10.10
	extension-header	

extension-header = HTTP-header

The extension-header mechanism allows additional Entity-Header fields to be defined without changing the protocol, but these fields cannot be assumed to be recognizable by the recipient. Unrecognized header fields should be ignored by the recipient and forwarded by proxies.

7.2 Entity Body

The entity body (if any) sent with an HTTP request or response is in a format and encoding defined by the Entity-Header fields.

Entity-Body = *OCTET

An entity body is included with a request message only when the request method calls for one. The presence of an entity body in a request is signaled by the inclusion of a Content-Length header field in the request message headers. HTTP/1.0 requests containing an entity body must include a valid Content-Length header field.

For response messages, whether or not an entity body is included with a message is dependent on both the request method and the response code. All responses to the HEAD request method must not include a body, even though the presence of entity header fields may lead one to believe they do. All 1xx (informational), 204 (no content), and 304 (not modified) responses must not include a body. All other responses must include an entity body or a Content-Length header field defined with a value of zero (0).

7.2.1 Type

When an Entity-Body is included with a message, the data type of that body is determined via the header fields Content-Type and Content-Encoding. These define a two-layer, ordered encoding model:

entity-body := Content-Encoding(Content-Type(data))

A Content-Type specifies the media type of the underlying data. A Content-Encoding may be used to indicate any additional content coding applied to the type, usually for the purpose of data compression, that is a property of the resource requested. The default for the content encoding is none (i.e., the identity function).

Any HTTP/1.0 message containing an entity body should include a Content-Type header field defining the media type of that body. If and only if the media type is not given by a Content-Type header, as is the case for Simple-Response messages, the recipient may attempt to guess the media type via inspection of its content and/or the name extension(s) of the URL used to identify the resource. If the media type remains unknown, the recipient should treat it as type "application/octet-stream".

7.2.2 Length

When an Entity-Body is included with a message, the length of that body may be determined in one of two ways. If a Content-Length header field is present, its value in bytes represents the length of the Entity-Body. Otherwise, the body length is determined by the closing of the connection by the server.

Closing the connection cannot be used to indicate the end of a request body, since it leaves no possibility for the server to send back a response. Therefore, HTTP/1.0 requests containing an entity body must include a valid Content-Length header field. If a request contains an entity body and Content-Length is not specified, and the server does not recognize or cannot calculate the length from other fields, then the server should send a 400 (bad request) response.

Note: Some older servers supply an invalid Content-Length when sending a document that contains server-side includes dynamically inserted into the data stream. It must be emphasized that this will not be tolerated by future versions of HTTP. Unless the client knows that it is receiving a response from a compliant server, it should not depend on the Content-Length value being correct.

8. Method Definitions

The set of common methods for HTTP/1.0 is defined below. Although this set can be expanded, additional methods cannot be assumed to share the same semantics for separately extended clients and servers.

8.1 GET

The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

The semantics of the GET method changes to a "conditional GET" if the request message includes an If-Modified-Since header field. A conditional GET method requests that the identified resource be transferred only if it has been modified since the date given by the If-Modified-Since header, as described in Section 10.9. The conditional GET method is intended to reduce network usage by allowing cached entities to be refreshed without requiring multiple requests or transferring unnecessary data.

8.2 HEAD

The HEAD method is identical to GET except that the server must not return any Entity-Body in the response. The metainformation contained in the HTTP headers in response to a HEAD request should be identical to the information sent in response to a GET request. This method can be used for obtaining metainformation about the resource identified by the Request-URI without transferring the Entity-Body itself. This method is often used for testing hypertext links for validity, accessibility, and recent modification.

There is no "conditional HEAD" request analogous to the conditional GET. If an If-Modified-Since header field is included with a HEAD request, it should be ignored.

8.3 POST

The POST method is used to request that the destination server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line. POST is designed to allow a uniform method to cover the following functions:

- o Annotation of existing resources;
- o Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles;
- o Providing a block of data, such as the result of submitting a form [3], to a data-handling process;
- o Extending a database through an append operation.

The actual function performed by the POST method is determined by the server and is usually dependent on the Request-URI. The posted entity is subordinate to that URI in the same way that a file is subordinate to a directory containing it, a news article is subordinate to a newsgroup to which it is posted, or a record is subordinate to a database.

A successful POST does not require that the entity be created as a resource on the origin server or made accessible for future reference. That is, the action performed by the POST method might not result in a resource that can be identified by a URI. In this case, either 200 (ok) or 204 (no content) is the appropriate response status, depending on whether or not the response includes an entity that describes the result.

If a resource has been created on the origin server, the response should be 201 (created) and contain an entity (preferably of type "text/html") which describes the status of the request and refers to the new resource.

A valid Content-Length is required on all HTTP/1.0 POST requests. An HTTP/1.0 server should respond with a 400 (bad request) message if it cannot determine the length of the request message's content.

Applications must not cache responses to a POST request because the application has no way of knowing that the server would return an equivalent response on some future request.

9. Status Code Definitions

Each Status-Code is described below, including a description of which method(s) it can follow and any metainformation required in the response.

9.1 Informational 1xx

This class of status code indicates a provisional response, consisting only of the Status-Line and optional headers, and is terminated by an empty line. HTTP/1.0 does not define any 1xx status codes and they are not a valid response to a HTTP/1.0 request. However, they may be useful for experimental applications which are outside the scope of this specification.

9.2 Successful 2xx

This class of status code indicates that the client's request was successfully received, understood, and accepted.

200 OK

The request has succeeded. The information returned with the response is dependent on the method used in the request, as follows:

GET an entity corresponding to the requested resource is sent in the response;

HEAD the response must only contain the header information and no Entity-Body;

POST an entity describing or containing the result of the action.

201 Created

The request has been fulfilled and resulted in a new resource being created. The newly created resource can be referenced by the URI(s) returned in the entity of the response. The origin server should create the resource before using this Status-Code. If the action cannot be carried out immediately, the server must include in the response body a description of when the resource will be available; otherwise, the server should respond with 202 (accepted).

Of the methods defined by this specification, only POST can create a resource.

202 Accepted

The request has been accepted for processing, but the processing has not been completed. The request may or may not eventually be acted upon, as it may be disallowed when processing actually takes place. There is no facility for re-sending a status code from an asynchronous operation such as this.

The 202 response is intentionally non-committal. Its purpose is to allow a server to accept a request for some other process (perhaps a batch-oriented process that is only run once per day) without requiring that the user agent's connection to the server persist until the process is completed. The entity returned with this response should include an indication of the request's current status and either a pointer to a status monitor or some estimate of when the user can expect the request to be fulfilled.

204 No Content

The server has fulfilled the request but there is no new information to send back. If the client is a user agent, it should not change its document view from that which caused the request to

be generated. This response is primarily intended to allow input for scripts or other actions to take place without causing a change to the user agent's active document view. The response may include new metainformation in the form of entity headers, which should apply to the document currently in the user agent's active view.

9.3 Redirection 3xx

This class of status code indicates that further action needs to be taken by the user agent in order to fulfill the request. The action required may be carried out by the user agent without interaction with the user if and only if the method used in the subsequent request is GET or HEAD. A user agent should never automatically redirect a request more than 5 times, since such redirections usually indicate an infinite loop.

300 Multiple Choices

This response code is not directly used by HTTP/1.0 applications, but serves as the default for interpreting the 3xx class of responses.

The requested resource is available at one or more locations. Unless it was a HEAD request, the response should include an entity containing a list of resource characteristics and locations from which the user or user agent can choose the one most appropriate. If the server has a preferred choice, it should include the URL in a Location field; user agents may use this field value for automatic redirection.

301 Moved Permanently

The requested resource has been assigned a new permanent URL and any future references to this resource should be done using that URL. Clients with link editing capabilities should automatically relink references to the Request-URI to the new reference returned by the server, where possible.

The new URL must be given by the Location field in the response. Unless it was a HEAD request, the Entity-Body of the response should contain a short note with a hyperlink to the new URL.

If the 301 status code is received in response to a request using the POST method, the user agent must not automatically redirect the request unless it can be confirmed by the user, since this might change the conditions under which the request was issued.

Note: When automatically redirecting a POST request after receiving a 301 status code, some existing user agents will erroneously change it into a GET request.

302 Moved Temporarily

The requested resource resides temporarily under a different URL. Since the redirection may be altered on occasion, the client should continue to use the Request-URI for future requests.

The URL must be given by the Location field in the response. Unless it was a HEAD request, the Entity-Body of the response should contain a short note with a hyperlink to the new URI(s).

If the 302 status code is received in response to a request using the POST method, the user agent must not automatically redirect the request unless it can be confirmed by the user, since this might change the conditions under which the request was issued.

Note: When automatically redirecting a POST request after receiving a 302 status code, some existing user agents will erroneously change it into a GET request.

304 Not Modified

If the client has performed a conditional GET request and access is allowed, but the document has not been modified since the date and time specified in the If-Modified-Since field, the server must respond with this status code and not send an Entity-Body to the client. Header fields contained in the response should only include information which is relevant to cache managers or which may have changed independently of the entity's Last-Modified date. Examples of relevant header fields include: Date, Server, and Expires. A cache should update its cached entity to reflect any new field values given in the 304 response.

9.4 Client Error 4xx

The 4xx class of status code is intended for cases in which the client seems to have erred. If the client has not completed the request when a 4xx code is received, it should immediately cease sending data to the server. Except when responding to a HEAD request, the server should include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition. These status codes are applicable to any request method.

Note: If the client is sending data, server implementations on TCP should be careful to ensure that the client acknowledges receipt of the packet(s) containing the response prior to closing the input connection. If the client continues sending data to the server after the close, the server's controller will send a reset packet to the client, which may erase the client's unacknowledged input buffers before they can be read and interpreted by the HTTP application.

400 Bad Request

The request could not be understood by the server due to malformed syntax. The client should not repeat the request without modifications.

401 Unauthorized

The request requires user authentication. The response must include a WWW-Authenticate header field (Section 10.16) containing a challenge applicable to the requested resource. The client may repeat the request with a suitable Authorization header field (Section 10.2). If the request already included Authorization credentials, then the 401 response indicates that authorization has been refused for those credentials. If the 401 response contains the same challenge as the prior response, and the user agent has already attempted authentication at least once, then the user should be presented the entity that was given in the response, since that entity may include relevant diagnostic information. HTTP access authentication is explained in Section 11.

403 Forbidden

The server understood the request, but is refusing to fulfill it. Authorization will not help and the request should not be repeated. If the request method was not HEAD and the server wishes to make public why the request has not been fulfilled, it should describe the reason for the refusal in the entity body. This status code is commonly used when the server does not wish to reveal exactly why the request has been refused, or when no other response is applicable.

404 Not Found

The server has not found anything matching the Request-URI. No indication is given of whether the condition is temporary or permanent. If the server does not wish to make this information available to the client, the status code 403 (forbidden) can be used instead.

9.5 Server Error 5xx

Response status codes beginning with the digit "5" indicate cases in which the server is aware that it has erred or is incapable of performing the request. If the client has not completed the request when a 5xx code is received, it should immediately cease sending data to the server. Except when responding to a HEAD request, the server should include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition. These response codes are applicable to any request method and there are no required header fields.

500 Internal Server Error

The server encountered an unexpected condition which prevented it from fulfilling the request.

501 Not Implemented

The server does not support the functionality required to fulfill the request. This is the appropriate response when the server does not recognize the request method and is not capable of supporting it for any resource.

502 Bad Gateway

The server, while acting as a gateway or proxy, received an invalid response from the upstream server it accessed in attempting to fulfill the request.

503 Service Unavailable

The server is currently unable to handle the request due to a temporary overloading or maintenance of the server. The implication is that this is a temporary condition which will be alleviated after some delay.

Note: The existence of the 503 status code does not imply that a server must use it when becoming overloaded. Some servers may wish to simply refuse the connection.

10. Header Field Definitions

This section defines the syntax and semantics of all commonly used HTTP/1.0 header fields. For general and entity header fields, both sender and recipient refer to either the client or the server, depending on who sends and who receives the message.

10.1 Allow

The Allow entity-header field lists the set of methods supported by the resource identified by the Request-URI. The purpose of this field is strictly to inform the recipient of valid methods associated with the resource. The Allow header field is not permitted in a request using the POST method, and thus should be ignored if it is received as part of a POST entity.

Allow = "Allow" ":" 1#method

Example of use:

Allow: GET, HEAD

This field cannot prevent a client from trying other methods. However, the indications given by the Allow header field value should be followed. The actual set of allowed methods is defined by the origin server at the time of each request.

A proxy must not modify the Allow header field even if it does not understand all the methods specified, since the user agent may have other means of communicating with the origin server.

The Allow header field does not indicate what methods are implemented by the server.

10.2 Authorization

A user agent that wishes to authenticate itself with a server--usually, but not necessarily, after receiving a 401 response--may do so by including an Authorization request-header field with the request. The Authorization field value consists of credentials containing the authentication information of the user agent for the realm of the resource being requested.

Authorization = "Authorization" ":" credentials

HTTP access authentication is described in Section 11. If a request is authenticated and a realm specified, the same credentials should be valid for all other requests within this realm.

Responses to requests containing an Authorization field are not cachable.

10.3 Content-Encoding

The Content-Encoding entity-header field is used as a modifier to the media-type. When present, its value indicates what additional content coding has been applied to the resource, and thus what decoding mechanism must be applied in order to obtain the media-type referenced by the Content-Type header field. The Content-Encoding is primarily used to allow a document to be compressed without losing the identity of its underlying media type.

Content-Encoding = "Content-Encoding" ":" content-coding

Content codings are defined in Section 3.5. An example of its use is

Content-Encoding: x-gzip

The Content-Encoding is a characteristic of the resource identified by the Request-URI. Typically, the resource is stored with this encoding and is only decoded before rendering or analogous usage.

10.4 Content-Length

The Content-Length entity-header field indicates the size of the Entity-Body, in decimal number of octets, sent to the recipient or, in the case of the HEAD method, the size of the Entity-Body that would have been sent had the request been a GET.

Content-Length = "Content-Length" ":" 1*DIGIT

An example is

Content-Length: 3495

Applications should use this field to indicate the size of the Entity-Body to be transferred, regardless of the media type of the entity. A valid Content-Length field value is required on all HTTP/1.0 request messages containing an entity body.

Any Content-Length greater than or equal to zero is a valid value. Section 7.2.2 describes how to determine the length of a response entity body if a Content-Length is not given.

Note: The meaning of this field is significantly different from the corresponding definition in MIME, where it is an optional field used within the "message/external-body" content-type. In HTTP, it should be used whenever the entity's length can be determined prior to being transferred.

10.5 Content-Type

The Content-Type entity-header field indicates the media type of the Entity-Body sent to the recipient or, in the case of the HEAD method, the media type that would have been sent had the request been a GET.

Content-Type = "Content-Type" ":" media-type

Media types are defined in Section 3.6. An example of the field is

Content-Type: text/html

Further discussion of methods for identifying the media type of an entity is provided in Section 7.2.1.

10.6 Date

The Date general-header field represents the date and time at which the message was originated, having the same semantics as orig-date in RFC 822. The field value is an HTTP-date, as described in Section 3.3.

Date = "Date" ":" HTTP-date

An example is

Date: Tue, 15 Nov 1994 08:12:31 GMT

If a message is received via direct connection with the user agent (in the case of requests) or the origin server (in the case of responses), then the date can be assumed to be the current date at the receiving end. However, since the date--as it is believed by the origin--is important for evaluating cached responses, origin servers should always include a Date header. Clients should only send a Date header field in messages that include an entity body, as in the case of the POST request, and even then it is optional. A received message which does not have a Date header field should be assigned one by the recipient if the message will be cached by that recipient or gatewayed via a protocol which requires a Date.

In theory, the date should represent the moment just before the entity is generated. In practice, the date can be generated at any time during the message origination without affecting its semantic value.

Note: An earlier version of this document incorrectly specified that this field should contain the creation date of the enclosed Entity-Body. This has been changed to reflect actual (and proper)

usage.

10.7 Expires

The Expires entity-header field gives the date/time after which the entity should be considered stale. This allows information providers to suggest the volatility of the resource, or a date after which the information may no longer be valid. Applications must not cache this entity beyond the date given. The presence of an Expires field does not imply that the original resource will change or cease to exist at, before, or after that time. However, information providers that know or even suspect that a resource will change by a certain date should include an Expires header with that date. The format is an absolute date and time as defined by HTTP-date in Section 3.3.

Expires = "Expires" ":" HTTP-date

An example of its use is

Expires: Thu, 01 Dec 1994 16:00:00 GMT

If the date given is equal to or earlier than the value of the Date header, the recipient must not cache the enclosed entity. If a resource is dynamic by nature, as is the case with many data-producing processes, entities from that resource should be given an appropriate Expires value which reflects that dynamism.

The Expires field cannot be used to force a user agent to refresh its display or reload a resource; its semantics apply only to caching mechanisms, and such mechanisms need only check a resource's expiration status when a new request for that resource is initiated.

User agents often have history mechanisms, such as "Back" buttons and history lists, which can be used to redisplay an entity retrieved earlier in a session. By default, the Expires field does not apply to history mechanisms. If the entity is still in storage, a history mechanism should display it even if the entity has expired, unless the user has specifically configured the agent to refresh expired history documents.

Note: Applications are encouraged to be tolerant of bad or misinformed implementations of the Expires header. A value of zero (0) or an invalid date format should be considered equivalent to an "expires immediately." Although these values are not legitimate for HTTP/1.0, a robust implementation is always desirable.

10.8 From

The From request-header field, if given, should contain an Internet e-mail address for the human user who controls the requesting user agent. The address should be machine-usable, as defined by mailbox in RFC 822 [7] (as updated by RFC 1123 [6]):

From = "From" ":" mailbox

An example is:

From: webmaster@w3.org

This header field may be used for logging purposes and as a means for identifying the source of invalid or unwanted requests. It should not be used as an insecure form of access protection. The interpretation of this field is that the request is being performed on behalf of the person given, who accepts responsibility for the method performed. In particular, robot agents should include this header so that the person responsible for running the robot can be contacted if problems occur on the receiving end.

The Internet e-mail address in this field may be separate from the Internet host which issued the request. For example, when a request is passed through a proxy, the original issuer's address should be used.

Note: The client should not send the From header field without the user's approval, as it may conflict with the user's privacy interests or their site's security policy. It is strongly recommended that the user be able to disable, enable, and modify the value of this field at any time prior to a request.

10.9 If-Modified-Since

The If-Modified-Since request-header field is used with the GET method to make it conditional: if the requested resource has not been modified since the time specified in this field, a copy of the resource will not be returned from the server; instead, a 304 (not modified) response will be returned without any Entity-Body.

If-Modified-Since = "If-Modified-Since" ":" HTTP-date

An example of the field is:

If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT

A conditional GET method requests that the identified resource be transferred only if it has been modified since the date given by the If-Modified-Since header. The algorithm for determining this includes the following cases:

- a) If the request would normally result in anything other than a 200 (ok) status, or if the passed If-Modified-Since date is invalid, the response is exactly the same as for a normal GET. A date which is later than the server's current time is invalid.
- b) If the resource has been modified since the If-Modified-Since date, the response is exactly the same as for a normal GET.
- c) If the resource has not been modified since a valid If-Modified-Since date, the server shall return a 304 (not modified) response.

The purpose of this feature is to allow efficient updates of cached information with a minimum amount of transaction overhead.

10.10 Last-Modified

The Last-Modified entity-header field indicates the date and time at which the sender believes the resource was last modified. The exact semantics of this field are defined in terms of how the recipient should interpret it: if the recipient has a copy of this resource which is older than the date given by the Last-Modified field, that copy should be considered stale.

Last-Modified = "Last-Modified" ":" HTTP-date

An example of its use is

Last-Modified: Tue, 15 Nov 1994 12:45:26 GMT

The exact meaning of this header field depends on the implementation of the sender and the nature of the original resource. For files, it may be just the file system last-modified time. For entities with dynamically included parts, it may be the most recent of the set of last-modify times for its component parts. For database gateways, it may be the last-update timestamp of the record. For virtual objects, it may be the last time the internal state changed.

An origin server must not send a Last-Modified date which is later than the server's time of message origination. In such cases, where the resource's last modification would indicate some time in the

future, the server must replace that date with the message origination date.

10.11 Location

The Location response-header field defines the exact location of the resource that was identified by the Request-URI. For 3xx responses, the location must indicate the server's preferred URL for automatic redirection to the resource. Only one absolute URL is allowed.

Location = "Location" ":" absoluteURI

An example is

Location: http://www.w3.org/hypertext/WWW/NewLocation.html

10.12 Pragma

The Pragma general-header field is used to include implementation-specific directives that may apply to any recipient along the request/response chain. All pragma directives specify optional behavior from the viewpoint of the protocol; however, some systems may require that behavior be consistent with the directives.

Pragma = "Pragma" ":" 1#pragma-directive

pragma-directive = "no-cache" | extension-pragma
extension-pragma = token ["=" word]

When the "no-cache" directive is present in a request message, an application should forward the request toward the origin server even if it has a cached copy of what is being requested. This allows a client to insist upon receiving an authoritative response to its request. It also allows a client to refresh a cached copy which is known to be corrupted or stale.

Pragma directives must be passed through by a proxy or gateway application, regardless of their significance to that application, since the directives may be applicable to all recipients along the request/response chain. It is not possible to specify a pragma for a specific recipient; however, any pragma directive not relevant to a recipient should be ignored by that recipient.

10.13 Referer

The Referer request-header field allows the client to specify, for the server's benefit, the address (URI) of the resource from which the Request-URI was obtained. This allows a server to generate lists

of back-links to resources for interest, logging, optimized caching, etc. It also allows obsolete or mistyped links to be traced for maintenance. The Referer field must not be sent if the Request-URI was obtained from a source that does not have its own URI, such as input from the user keyboard.

Referer = "Referer" ":" (absoluteURI | relativeURI)

Example:

Referer: http://www.w3.org/hypertext/DataSources/Overview.html

If a partial URI is given, it should be interpreted relative to the Request-URI. The URI must not include a fragment.

Note: Because the source of a link may be private information or may reveal an otherwise private information source, it is strongly recommended that the user be able to select whether or not the Referer field is sent. For example, a browser client could have a toggle switch for browsing openly/anonymously, which would respectively enable/disable the sending of Referer and From information.

10.14 Server

The Server response-header field contains information about the software used by the origin server to handle the request. The field can contain multiple product tokens (Section 3.7) and comments identifying the server and any significant subproducts. By convention, the product tokens are listed in order of their significance for identifying the application.

Server = "Server" ":" 1*(product | comment)

Example:

Server: CERN/3.0 libwww/2.17

If the response is being forwarded through a proxy, the proxy application must not add its data to the product list.

Note: Revealing the specific software version of the server may allow the server machine to become more vulnerable to attacks against software that is known to contain security holes. Server implementors are encouraged to make this field a configurable option.

Note: Some existing servers fail to restrict themselves to the product token syntax within the Server field.

10.15 User-Agent

The User-Agent request-header field contains information about the user agent originating the request. This is for statistical purposes, the tracing of protocol violations, and automated recognition of user agents for the sake of tailoring responses to avoid particular user agent limitations. Although it is not required, user agents should include this field with requests. The field can contain multiple product tokens (Section 3.7) and comments identifying the agent and any subproducts which form a significant part of the user agent. By convention, the product tokens are listed in order of their significance for identifying the application.

User-Agent = "User-Agent" ":" 1*(product | comment)

Example:

User-Agent: CERN-LineMode/2.15 libwww/2.17b3

Note: Some current proxy applications append their product information to the list in the User-Agent field. This is not recommended, since it makes machine interpretation of these fields ambiguous.

Note: Some existing clients fail to restrict themselves to the product token syntax within the User-Agent field.

10.16 WWW-Authenticate

The WWW-Authenticate response-header field must be included in 401 (unauthorized) response messages. The field value consists of at least one challenge that indicates the authentication scheme(s) and parameters applicable to the Request-URI.

WWW-Authenticate = "WWW-Authenticate" ":" 1#challenge

The HTTP access authentication process is described in Section 11. User agents must take special care in parsing the WWW-Authenticate field value if it contains more than one challenge, or if more than one WWW-Authenticate header field is provided, since the contents of a challenge may itself contain a comma-separated list of authentication parameters.

11. Access Authentication

HTTP provides a simple challenge-response authentication mechanism which may be used by a server to challenge a client request and by a client to provide authentication information. It uses an extensible, case-insensitive token to identify the authentication scheme, followed by a comma-separated list of attribute-value pairs which carry the parameters necessary for achieving authentication via that scheme.

auth-scheme = token

auth-param = token "=" quoted-string

The 401 (unauthorized) response message is used by an origin server to challenge the authorization of a user agent. This response must include a WWW-Authenticate header field containing at least one challenge applicable to the requested resource.

challenge = auth-scheme 1*SP realm *("," auth-param)

realm = "realm" "=" realm-value
realm-value = quoted-string

The realm attribute (case-insensitive) is required for all authentication schemes which issue a challenge. The realm value (case-sensitive), in combination with the canonical root URL of the server being accessed, defines the protection space. These realms allow the protected resources on a server to be partitioned into a set of protection spaces, each with its own authentication scheme and/or authorization database. The realm value is a string, generally assigned by the origin server, which may have additional semantics specific to the authentication scheme.

A user agent that wishes to authenticate itself with a server--usually, but not necessarily, after receiving a 401 response--may do so by including an Authorization header field with the request. The Authorization field value consists of credentials containing the authentication information of the user agent for the realm of the resource being requested.

credentials = basic-credentials
| (auth-scheme #auth-param)

The domain over which credentials can be automatically applied by a user agent is determined by the protection space. If a prior request has been authorized, the same credentials may be reused for all other requests within that protection space for a period of time determined

by the authentication scheme, parameters, and/or user preference. Unless otherwise defined by the authentication scheme, a single protection space cannot extend outside the scope of its server.

If the server does not wish to accept the credentials sent with a request, it should return a 403 (forbidden) response.

The HTTP protocol does not restrict applications to this simple challenge-response mechanism for access authentication. Additional mechanisms may be used, such as encryption at the transport level or via message encapsulation, and with additional header fields specifying authentication information. However, these additional mechanisms are not defined by this specification.

Proxies must be completely transparent regarding user agent authentication. That is, they must forward the WWW-Authenticate and Authorization headers untouched, and must not cache the response to a request containing Authorization. HTTP/1.0 does not provide a means for a client to be authenticated with a proxy.

11.1 Basic Authentication Scheme

The "basic" authentication scheme is based on the model that the user agent must authenticate itself with a user-ID and a password for each realm. The realm value should be considered an opaque string which can only be compared for equality with other realms on that server. The server will authorize the request only if it can validate the user-ID and password for the protection space of the Request-URI. There are no optional authentication parameters.

Upon receipt of an unauthorized request for a URI within the protection space, the server should respond with a challenge like the following:

WWW-Authenticate: Basic realm="WallyWorld"

where "WallyWorld" is the string assigned by the server to identify the protection space of the Request-URI.

To receive authorization, the client sends the user-ID and password, separated by a single colon (":") character, within a base64 [5] encoded string in the credentials.

basic-credentials = "Basic" SP basic-cookie

basic-cookie = <base64 [5] encoding of userid-password,
except not limited to 76 char/line>

userid-password = [token] ":" *TEXT

If the user agent wishes to send the user-ID "Aladdin" and password "open sesame", it would use the following header field:

Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==

The basic authentication scheme is a non-secure method of filtering unauthorized access to resources on an HTTP server. It is based on the assumption that the connection between the client and the server can be regarded as a trusted carrier. As this is not generally true on an open network, the basic authentication scheme should be used accordingly. In spite of this, clients should implement the scheme in order to communicate with servers that use it.

12. Security Considerations

This section is meant to inform application developers, information providers, and users of the security limitations in HTTP/1.0 as described by this document. The discussion does not include definitive solutions to the problems revealed, though it does make some suggestions for reducing security risks.

12.1 Authentication of Clients

As mentioned in Section 11.1, the Basic authentication scheme is not a secure method of user authentication, nor does it prevent the Entity-Body from being transmitted in clear text across the physical network used as the carrier. HTTP/1.0 does not prevent additional authentication schemes and encryption mechanisms from being employed to increase security.

12.2 Safe Methods

The writers of client software should be aware that the software represents the user in their interactions over the Internet, and should be careful to allow the user to be aware of any actions they may take which may have an unexpected significance to themselves or others.

In particular, the convention has been established that the GET and HEAD methods should never have the significance of taking an action other than retrieval. These methods should be considered "safe." This allows user agents to represent other methods, such as POST, in a special way, so that the user is made aware of the fact that a possibly unsafe action is being requested.

Naturally, it is not possible to ensure that the server does not generate side-effects as a result of performing a GET request; in fact, some dynamic resources consider that a feature. The important distinction here is that the user did not request the side-effects, so therefore cannot be held accountable for them.

12.3 Abuse of Server Log Information

A server is in the position to save personal data about a user's requests which may identify their reading patterns or subjects of interest. This information is clearly confidential in nature and its handling may be constrained by law in certain countries. People using the HTTP protocol to provide data are responsible for ensuring that such material is not distributed without the permission of any individuals that are identifiable by the published results.

12.4 Transfer of Sensitive Information

Like any generic data transfer protocol, HTTP cannot regulate the content of the data that is transferred, nor is there any a priori method of determining the sensitivity of any particular piece of information within the context of any given request. Therefore, applications should supply as much control over this information as possible to the provider of that information. Three header fields are worth special mention in this context: Server, Referer and From.

Revealing the specific software version of the server may allow the server machine to become more vulnerable to attacks against software that is known to contain security holes. Implementors should make the Server header field a configurable option.

The Referer field allows reading patterns to be studied and reverse links drawn. Although it can be very useful, its power can be abused if user details are not separated from the information contained in the Referer. Even when the personal information has been removed, the Referer field may indicate a private document's URI whose publication would be inappropriate.

The information sent in the From field might conflict with the user's privacy interests or their site's security policy, and hence it should not be transmitted without the user being able to disable, enable, and modify the contents of the field. The user must be able to set the contents of this field within a user preference or application defaults configuration.

We suggest, though do not require, that a convenient toggle interface be provided for the user to enable or disable the sending of From and Referer information.

12.5 Attacks Based On File and Path Names

Implementations of HTTP origin servers should be careful to restrict the documents returned by HTTP requests to be only those that were intended by the server administrators. If an HTTP server translates HTTP URIs directly into file system calls, the server must take special care not to serve files that were not intended to be delivered to HTTP clients. For example, Unix, Microsoft Windows, and other operating systems use "." as a path component to indicate a directory level above the current one. On such a system, an HTTP server must disallow any such construct in the Request-URI if it would otherwise allow access to a resource outside those intended to be accessible via the HTTP server. Similarly, files intended for reference only internally to the server (such as access control files, configuration files, and script code) must be protected from inappropriate retrieval, since they might contain sensitive information. Experience has shown that minor bugs in such HTTP server implementations have turned into security risks.

13. Acknowledgments

This specification makes heavy use of the augmented BNF and generic constructs defined by David H. Crocker for RFC 822 [7]. Similarly, it reuses many of the definitions provided by Nathaniel Borenstein and Ned Freed for MIME [5]. We hope that their inclusion in this specification will help reduce past confusion over the relationship between HTTP/1.0 and Internet mail message formats.

The HTTP protocol has evolved considerably over the past four years. It has benefited from a large and active developer community--the many people who have participated on the www-talk mailing list--and it is that community which has been most responsible for the success of HTTP and of the World-Wide Web in general. Marc Andreessen, Robert Cailliau, Daniel W. Connolly, Bob Denny, Jean-Francois Groff, Phillip M. Hallam-Baker, Hakon W. Lie, Ari Luotonen, Rob McCool, Lou Montulli, Dave Raggett, Tony Sanders, and Marc VanHeyningen deserve special recognition for their efforts in defining aspects of the protocol for early versions of this specification.

Paul Hoffman contributed sections regarding the informational status of this document and Appendices C and D.

This document has benefited greatly from the comments of all those participating in the HTTP-WG. In addition to those already mentioned, the following individuals have contributed to this specification:

Gary Adams	Harald Tveit Alvestrand
Keith Ball	Brian Behlendorf
Paul Burchard	Maurizio Codogno
Mike Cowlishaw	Roman Czyborra
Michael A. Dolan	John Franks
Jim Gettys	Marc Hedlund
Koen Holtman	Alex Hopmann
Bob Jernigan	Shel Kaphan
Martijn Koster	Dave Kristol
Daniel LaLiberte	Paul Leach
Albert Lunde	John C. Mallery
Larry Masinter	Mitra
Jeffrey Mogul	Gavin Nicol
Bill Perry	Jeffrey Perry
Owen Rees	Luigi Rizzo
David Robinson	Marc Salomon
Rich Salz	Jim Seidman
Chuck Shotton	Eric W. Sink
Simon E. Speró	Robert S. Thau
Francois Yergeau	Mary Ellen Zurko
Jean-Philippe Martin-Flatin	

14. References

- [1] Anklesaria, F., McCahill, M., Lindner, P., Johnson, D., Torrey, D., and B. Alberti, "The Internet Gopher Protocol: A Distributed Document Search and Retrieval Protocol", RFC 1436, University of Minnesota, March 1993.
- [2] Berners-Lee, T., "Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web", RFC 1630, CERN, June 1994.
- [3] Berners-Lee, T., and D. Connolly, "Hypertext Markup Language - 2.0", RFC 1866, MIT/W3C, November 1995.
- [4] Berners-Lee, T., Masinter, L., and M. McCahill, "Uniform Resource Locators (URL)", RFC 1738, CERN, Xerox PARC, University of Minnesota, December 1994.

- [5] Borenstein, N., and N. Freed, "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies", RFC 1521, Bellcore, Innosoft, September 1993.
- [6] Braden, R., "Requirements for Internet hosts - Application and Support", STD 3, RFC 1123, IETF, October 1989.
- [7] Crocker, D., "Standard for the Format of ARPA Internet Text Messages", STD 11, RFC 822, UDEL, August 1982.
- [8] F. Davis, B. Kahle, H. Morris, J. Salem, T. Shen, R. Wang, J. Sui, and M. Grinbaum. "WAIS Interface Protocol Prototype Functional Specification." (v1.5), Thinking Machines Corporation, April 1990.
- [9] Fielding, R., "Relative Uniform Resource Locators", RFC 1808, UC Irvine, June 1995.
- [10] Horton, M., and R. Adams, "Standard for interchange of USENET Messages", RFC 1036 (Obsoletes RFC 850), AT&T Bell Laboratories, Center for Seismic Studies, December 1987.
- [11] Kantor, B., and P. Lapsley, "Network News Transfer Protocol: A Proposed Standard for the Stream-Based Transmission of News", RFC 977, UC San Diego, UC Berkeley, February 1986.
- [12] Postel, J., "Simple Mail Transfer Protocol." STD 10, RFC 821, USC/ISI, August 1982.
- [13] Postel, J., "Media Type Registration Procedure." RFC 1590, USC/ISI, March 1994.
- [14] Postel, J., and J. Reynolds, "File Transfer Protocol (FTP)", STD 9, RFC 959, USC/ISI, October 1985.
- [15] Reynolds, J., and J. Postel, "Assigned Numbers", STD 2, RFC 1700, USC/ISI, October 1994.
- [16] Sollins, K., and L. Masinter, "Functional Requirements for Uniform Resource Names", RFC 1737, MIT/LCS, Xerox Corporation, December 1994.
- [17] US-ASCII. Coded Character Set - 7-Bit American Standard Code for Information Interchange. Standard ANSI X3.4-1986, ANSI, 1986.

- [18] ISO-8859. International Standard -- Information Processing --
8-bit Single-Byte Coded Graphic Character Sets --
Part 1: Latin alphabet No. 1, ISO 8859-1:1987.
Part 2: Latin alphabet No. 2, ISO 8859-2, 1987.
Part 3: Latin alphabet No. 3, ISO 8859-3, 1988.
Part 4: Latin alphabet No. 4, ISO 8859-4, 1988.
Part 5: Latin/Cyrillic alphabet, ISO 8859-5, 1988.
Part 6: Latin/Arabic alphabet, ISO 8859-6, 1987.
Part 7: Latin/Greek alphabet, ISO 8859-7, 1987.
Part 8: Latin/Hebrew alphabet, ISO 8859-8, 1988.
Part 9: Latin alphabet No. 5, ISO 8859-9, 1990.

15. Authors' Addresses

Tim Berners-Lee
Director, W3 Consortium
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, U.S.A.

Fax: +1 (617) 258 8682
EMail: timbl@w3.org

Roy T. Fielding
Department of Information and Computer Science
University of California
Irvine, CA 92717-3425, U.S.A.

Fax: +1 (714) 824-4056
EMail: fielding@ics.uci.edu

Henrik Frystyk Nielsen
W3 Consortium
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, U.S.A.

Fax: +1 (617) 258 8682
EMail: frystyk@w3.org

Appendices

These appendices are provided for informational reasons only -- they do not form a part of the HTTP/1.0 specification.

A. Internet Media Type message/http

In addition to defining the HTTP/1.0 protocol, this document serves as the specification for the Internet media type "message/http". The following is to be registered with IANA [13].

Media Type name: message
Media subtype name: http
Required parameters: none
Optional parameters: version, msgtype

version: The HTTP-Version number of the enclosed message (e.g., "1.0"). If not present, the version can be determined from the first line of the body.

msgtype: The message type -- "request" or "response". If not present, the type can be determined from the first line of the body.

Encoding considerations: only "7bit", "8bit", or "binary" are permitted

Security considerations: none

B. Tolerant Applications

Although this document specifies the requirements for the generation of HTTP/1.0 messages, not all applications will be correct in their implementation. We therefore recommend that operational applications be tolerant of deviations whenever those deviations can be interpreted unambiguously.

Clients should be tolerant in parsing the Status-Line and servers tolerant when parsing the Request-Line. In particular, they should accept any amount of SP or HT characters between fields, even though only a single SP is required.

The line terminator for HTTP-header fields is the sequence CRLF. However, we recommend that applications, when parsing such headers, recognize a single LF as a line terminator and ignore the leading CR.

C. Relationship to MIME

HTTP/1.0 uses many of the constructs defined for Internet Mail (RFC 822 [7]) and the Multipurpose Internet Mail Extensions (MIME [5]) to allow entities to be transmitted in an open variety of representations and with extensible mechanisms. However, RFC 1521 discusses mail, and HTTP has a few features that are different than those described in RFC 1521. These differences were carefully chosen to optimize performance over binary connections, to allow greater freedom in the use of new media types, to make date comparisons easier, and to acknowledge the practice of some early HTTP servers and clients.

At the time of this writing, it is expected that RFC 1521 will be revised. The revisions may include some of the practices found in HTTP/1.0 but not in RFC 1521.

This appendix describes specific areas where HTTP differs from RFC 1521. Proxies and gateways to strict MIME environments should be aware of these differences and provide the appropriate conversions where necessary. Proxies and gateways from MIME environments to HTTP also need to be aware of the differences because some conversions may be required.

C.1 Conversion to Canonical Form

RFC 1521 requires that an Internet mail entity be converted to canonical form prior to being transferred, as described in Appendix G of RFC 1521 [5]. Section 3.6.1 of this document describes the forms allowed for subtypes of the "text" media type when transmitted over HTTP.

RFC 1521 requires that content with a Content-Type of "text" represent line breaks as CRLF and forbids the use of CR or LF outside of line break sequences. HTTP allows CRLF, bare CR, and bare LF to indicate a line break within text content when a message is transmitted over HTTP.

Where it is possible, a proxy or gateway from HTTP to a strict RFC 1521 environment should translate all line breaks within the text media types described in Section 3.6.1 of this document to the RFC 1521 canonical form of CRLF. Note, however, that this may be complicated by the presence of a Content-Encoding and by the fact that HTTP allows the use of some character sets which do not use octets 13 and 10 to represent CR and LF, as is the case for some multi-byte character sets.

C.2 Conversion of Date Formats

HTTP/1.0 uses a restricted set of date formats (Section 3.3) to simplify the process of date comparison. Proxies and gateways from other protocols should ensure that any Date header field present in a message conforms to one of the HTTP/1.0 formats and rewrite the date if necessary.

C.3 Introduction of Content-Encoding

RFC 1521 does not include any concept equivalent to HTTP/1.0's Content-Encoding header field. Since this acts as a modifier on the media type, proxies and gateways from HTTP to MIME-compliant protocols must either change the value of the Content-Type header field or decode the Entity-Body before forwarding the message. (Some experimental applications of Content-Type for Internet mail have used a media-type parameter of ";conversion=<content-coding>" to perform an equivalent function as Content-Encoding. However, this parameter is not part of RFC 1521.)

C.4 No Content-Transfer-Encoding

HTTP does not use the Content-Transfer-Encoding (CTE) field of RFC 1521. Proxies and gateways from MIME-compliant protocols to HTTP must remove any non-identity CTE ("quoted-printable" or "base64") encoding prior to delivering the response message to an HTTP client.

Proxies and gateways from HTTP to MIME-compliant protocols are responsible for ensuring that the message is in the correct format and encoding for safe transport on that protocol, where "safe transport" is defined by the limitations of the protocol being used. Such a proxy or gateway should label the data with an appropriate Content-Transfer-Encoding if doing so will improve the likelihood of safe transport over the destination protocol.

C.5 HTTP Header Fields in Multipart Body-Parts

In RFC 1521, most header fields in multipart body-parts are generally ignored unless the field name begins with "Content-". In HTTP/1.0, multipart body-parts may contain any HTTP header fields which are significant to the meaning of that part.

D. Additional Features

This appendix documents protocol elements used by some existing HTTP implementations, but not consistently and correctly across most HTTP/1.0 applications. Implementors should be aware of these features, but cannot rely upon their presence in, or interoperability

with, other HTTP/1.0 applications.

D.1 Additional Request Methods

D.1.1 PUT

The PUT method requests that the enclosed entity be stored under the supplied Request-URI. If the Request-URI refers to an already existing resource, the enclosed entity should be considered as a modified version of the one residing on the origin server. If the Request-URI does not point to an existing resource, and that URI is capable of being defined as a new resource by the requesting user agent, the origin server can create the resource with that URI.

The fundamental difference between the POST and PUT requests is reflected in the different meaning of the Request-URI. The URI in a POST request identifies the resource that will handle the enclosed entity as data to be processed. That resource may be a data-accepting process, a gateway to some other protocol, or a separate entity that accepts annotations. In contrast, the URI in a PUT request identifies the entity enclosed with the request -- the user agent knows what URI is intended and the server should not apply the request to some other resource.

D.1.2 DELETE

The DELETE method requests that the origin server delete the resource identified by the Request-URI.

D.1.3 LINK

The LINK method establishes one or more Link relationships between the existing resource identified by the Request-URI and other existing resources.

D.1.4 UNLINK

The UNLINK method removes one or more Link relationships from the existing resource identified by the Request-URI.

D.2 Additional Header Field Definitions

D.2.1 Accept

The Accept request-header field can be used to indicate a list of media ranges which are acceptable as a response to the request. The asterisk "*" character is used to group media types into ranges, with "*/*" indicating all media types and "type/*" indicating all subtypes

of that type. The set of ranges given by the client should represent what types are acceptable given the context of the request.

D.2.2 Accept-Charset

The Accept-Charset request-header field can be used to indicate a list of preferred character sets other than the default US-ASCII and ISO-8859-1. This field allows clients capable of understanding more comprehensive or special-purpose character sets to signal that capability to a server which is capable of representing documents in those character sets.

D.2.3 Accept-Encoding

The Accept-Encoding request-header field is similar to Accept, but restricts the content-coding values which are acceptable in the response.

D.2.4 Accept-Language

The Accept-Language request-header field is similar to Accept, but restricts the set of natural languages that are preferred as a response to the request.

D.2.5 Content-Language

The Content-Language entity-header field describes the natural language(s) of the intended audience for the enclosed entity. Note that this may not be equivalent to all the languages used within the entity.

D.2.6 Link

The Link entity-header field provides a means for describing a relationship between the entity and some other resource. An entity may include multiple Link values. Links at the metainformation level typically indicate relationships like hierarchical structure and navigation paths.

D.2.7 MIME-Version

HTTP messages may include a single MIME-Version general-header field to indicate what version of the MIME protocol was used to construct the message. Use of the MIME-Version header field, as defined by RFC 1521 [5], should indicate that the message is MIME-conformant. Unfortunately, some older HTTP/1.0 servers send it indiscriminately, and thus this field should be ignored.

D.2.8 Retry-After

The Retry-After response-header field can be used with a 503 (service unavailable) response to indicate how long the service is expected to be unavailable to the requesting client. The value of this field can be either an HTTP-date or an integer number of seconds (in decimal) after the time of the response.

D.2.9 Title

The Title entity-header field indicates the title of the entity.

D.2.10 URI

The URI entity-header field may contain some or all of the Uniform Resource Identifiers (Section 3.2) by which the Request-URI resource can be identified. There is no guarantee that the resource can be accessed using the URI(s) specified.

Network Working Group
Request for Comments: 2068
Category: Standards Track

R. Fielding
UC Irvine
J. Gettys
J. Mogul
DEC
H. Frystyk
T. Berners-Lee
MIT/LCS
January 1997

JTSB-508-US
(7)

Hypertext Transfer Protocol -- HTTP/1.1

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Abstract

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, object-oriented protocol which can be used for many tasks, such as name servers and distributed object management systems, through extension of its request methods. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred.

HTTP has been in use by the World-Wide Web global information initiative since 1990. This specification defines the protocol referred to as "HTTP/1.1".

Table of Contents

1 Introduction.....	7
1.1 Purpose	7
1.2 Requirements	7
1.3 Terminology	8
1.4 Overall Operation	11
2 Notational Conventions and Generic Grammar.....	13
2.1 Augmented BNF	13
2.2 Basic Rules	15
3 Protocol Parameters.....	17
3.1 HTTP Version	17

3.2 Uniform Resource Identifiers	18
3.2.1 General Syntax	18
3.2.2 http URL	19
3.2.3 URI Comparison	20
3.3 Date/Time Formats	21
3.3.1 Full Date	21
3.3.2 Delta Seconds	22
3.4 Character Sets	22
3.5 Content Codings	23
3.6 Transfer Codings	24
3.7 Media Types	25
3.7.1 Canonicalization and Text Defaults	26
3.7.2 Multipart Types	27
3.8 Product Tokens	28
3.9 Quality Values	28
3.10 Language Tags	28
3.11 Entity Tags	29
3.12 Range Units	30
4 HTTP Message.....	30
4.1 Message Types	30
4.2 Message Headers	31
4.3 Message Body	32
4.4 Message Length	32
4.5 General Header Fields	34
5 Request.....	34
5.1 Request-Line	34
5.1.1 Method	35
5.1.2 Request-URI	35
5.2 The Resource Identified by a Request	37
5.3 Request Header Fields	37
6 Response.....	38
6.1 Status-Line	38
6.1.1 Status Code and Reason Phrase	39
6.2 Response Header Fields	41
7 Entity.....	41
7.1 Entity Header Fields	41
7.2 Entity Body	42
7.2.1 Type	42
7.2.2 Length	43
8 Connections.....	43
8.1 Persistent Connections	43
8.1.1 Purpose	43
8.1.2 Overall Operation	44
8.1.3 Proxy Servers	45
8.1.4 Practical Considerations	45
8.2 Message Transmission Requirements	46
9 Method Definitions.....	48
9.1 Safe and Idempotent Methods	48

9.1.1 Safe Methods	48
9.1.2 Idempotent Methods	49
9.2 OPTIONS	49
9.3 GET	50
9.4 HEAD	50
9.5 POST	51
9.6 PUT	52
9.7 DELETE	53
9.8 TRACE	53
10 Status Code Definitions.....	53
10.1 Informational lxx	54
10.1.1 100 Continue	54
10.1.2 101 Switching Protocols	54
10.2 Successful 2xx	54
10.2.1 200 OK	54
10.2.2 201 Created	55
10.2.3 202 Accepted	55
10.2.4 203 Non-Authoritative Information	55
10.2.5 204 No Content	55
10.2.6 205 Reset Content	56
10.2.7 206 Partial Content	56
10.3 Redirection 3xx	56
10.3.1 300 Multiple Choices	57
10.3.2 301 Moved Permanently	57
10.3.3 302 Moved Temporarily	58
10.3.4 303 See Other	58
10.3.5 304 Not Modified	58
10.3.6 305 Use Proxy	59
10.4 Client Error 4xx	59
10.4.1 400 Bad Request	60
10.4.2 401 Unauthorized	60
10.4.3 402 Payment Required	60
10.4.4 403 Forbidden	60
10.4.5 404 Not Found	60
10.4.6 405 Method Not Allowed	61
10.4.7 406 Not Acceptable	61
10.4.8 407 Proxy Authentication Required	61
10.4.9 408 Request Timeout	62
10.4.10 409 Conflict	62
10.4.11 410 Gone	62
10.4.12 411 Length Required	63
10.4.13 412 Precondition Failed	63
10.4.14 413 Request Entity Too Large	63
10.4.15 414 Request-URI Too Long	63
10.4.16 415 Unsupported Media Type	63
10.5 Server Error 5xx	64
10.5.1 500 Internal Server Error	64
10.5.2 501 Not Implemented	64

10.5.3 502 Bad Gateway	64
10.5.4 503 Service Unavailable	64
10.5.5 504 Gateway Timeout	64
10.5.6 505 HTTP Version Not Supported	65
11 Access Authentication.....	65
11.1 Basic Authentication Scheme	66
11.2 Digest Authentication Scheme	67
12 Content Negotiation.....	67
12.1 Server-driven Negotiation	68
12.2 Agent-driven Negotiation	69
12.3 Transparent Negotiation	70
13 Caching in HTTP.....	70
13.1.1 Cache Correctness	72
13.1.2 Warnings	73
13.1.3 Cache-control Mechanisms	74
13.1.4 Explicit User Agent Warnings	74
13.1.5 Exceptions to the Rules and Warnings	75
13.1.6 Client-controlled Behavior	75
13.2 Expiration Model	75
13.2.1 Server-Specified Expiration	75
13.2.2 Heuristic Expiration	76
13.2.3 Age Calculations	77
13.2.4 Expiration Calculations	79
13.2.5 Disambiguating Expiration Values	80
13.2.6 Disambiguating Multiple Responses	80
13.3 Validation Model	81
13.3.1 Last-modified Dates	82
13.3.2 Entity Tag Cache Validators	82
13.3.3 Weak and Strong Validators	82
13.3.4 Rules for When to Use Entity Tags and Last- modified Dates.....	85
13.3.5 Non-validating Conditionals	86
13.4 Response Cachability	86
13.5 Constructing Responses From Caches	87
13.5.1 End-to-end and Hop-by-hop Headers	88
13.5.2 Non-modifiable Headers	88
13.5.3 Combining Headers	89
13.5.4 Combining Byte Ranges	90
13.6 Caching Negotiated Responses	90
13.7 Shared and Non-Shared Caches	91
13.8 Errors or Incomplete Response Cache Behavior	91
13.9 Side Effects of GET and HEAD	92
13.10 Invalidation After Updates or Deletions	92
13.11 Write-Through Mandatory	93
13.12 Cache Replacement	93
13.13 History Lists	93
14 Header Field Definitions.....	94
14.1 Accept	95

14.2 Accept-Charset	97
14.3 Accept-Encoding	97
14.4 Accept-Language	98
14.5 Accept-Ranges	99
14.6 Age	99
14.7 Allow	100
14.8 Authorization	100
14.9 Cache-Control	101
14.9.1 What is Cachable	103
14.9.2 What May be Stored by Caches	103
14.9.3 Modifications of the Basic Expiration Mechanism	104
14.9.4 Cache Revalidation and Reload Controls	105
14.9.5 No-Transform Directive	107
14.9.6 Cache Control Extensions	108
14.10 Connection	109
14.11 Content-Base	109
14.12 Content-Encoding	110
14.13 Content-Language	110
14.14 Content-Length	111
14.15 Content-Location	112
14.16 Content-MD5	113
14.17 Content-Range	114
14.18 Content-Type	116
14.19 Date	116
14.20 ETag	117
14.21 Expires	117
14.22 From	118
14.23 Host	119
14.24 If-Modified-Since	119
14.25 If-Match	121
14.26 If-None-Match	122
14.27 If-Range	123
14.28 If-Unmodified-Since	124
14.29 Last-Modified	124
14.30 Location	125
14.31 Max-Forwards	125
14.32 Pragma	126
14.33 Proxy-Authenticate	127
14.34 Proxy-Authorization	127
14.35 Public	127
14.36 Range	128
14.36.1 Byte Ranges	128
14.36.2 Range Retrieval Requests	130
14.37 Referer	131
14.38 Retry-After	131
14.39 Server	132
14.40 Transfer-Encoding	132
14.41 Upgrade	132

14.42 User-Agent	134
14.43 Vary	134
14.44 Via	135
14.45 Warning	137
14.46 WWW-Authenticate	139
15 Security Considerations.....	139
15.1 Authentication of Clients	139
15.2 Offering a Choice of Authentication Schemes	140
15.3 Abuse of Server Log Information	141
15.4 Transfer of Sensitive Information	141
15.5 Attacks Based On File and Path Names	142
15.6 Personal Information	143
15.7 Privacy Issues Connected to Accept Headers	143
15.8 DNS Spoofing	144
15.9 Location Headers and Spoofing	144
16 Acknowledgments.....	144
17 References.....	146
18 Authors' Addresses.....	149
19 Appendices.....	150
19.1 Internet Media Type message/http	150
19.2 Internet Media Type multipart/byteranges	150
19.3 Tolerant Applications	151
19.4 Differences Between HTTP Entities and MIME Entities.....	152
19.4.1 Conversion to Canonical Form	152
19.4.2 Conversion of Date Formats	153
19.4.3 Introduction of Content-Encoding	153
19.4.4 No Content-Transfer-Encoding	153
19.4.5 HTTP Header Fields in Multipart Body-Parts	153
19.4.6 Introduction of Transfer-Encoding	154
19.4.7 MIME-Version	154
19.5 Changes from HTTP/1.0	154
19.5.1 Changes to Simplify Multi-homed Web Servers and Conserve IP Addresses	155
19.6 Additional Features	156
19.6.1 Additional Request Methods	156
19.6.2 Additional Header Field Definitions	156
19.7 Compatibility with Previous Versions	160
19.7.1 Compatibility with HTTP/1.0 Persistent Connections.....	161

1 Introduction

1.1 Purpose

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. HTTP has been in use by the World-Wide Web global information initiative since 1990. The first version of HTTP, referred to as HTTP/0.9, was a simple protocol for raw data transfer across the Internet. HTTP/1.0, as defined by RFC 1945 [6], improved the protocol by allowing messages to be in the format of MIME-like messages, containing meta-information about the data transferred and modifiers on the request/response semantics. However, HTTP/1.0 does not sufficiently take into consideration the effects of hierarchical proxies, caching, the need for persistent connections, and virtual hosts. In addition, the proliferation of incompletely-implemented applications calling themselves "HTTP/1.0" has necessitated a protocol version change in order for two communicating applications to determine each other's true capabilities.

This specification defines the protocol referred to as "HTTP/1.1". This protocol includes more stringent requirements than HTTP/1.0 in order to ensure reliable implementation of its features.

Practical information systems require more functionality than simple retrieval, including search, front-end update, and annotation. HTTP allows an open-ended set of methods that indicate the purpose of a request. It builds on the discipline of reference provided by the Uniform Resource Identifier (URI) [3][20], as a location (URL) [4] or name (URN) , for indicating the resource to which a method is to be applied. Messages are passed in a format similar to that used by Internet mail as defined by the Multipurpose Internet Mail Extensions (MIME).

HTTP is also used as a generic protocol for communication between user agents and proxies/gateways to other Internet systems, including those supported by the SMTP [16], NNTP [13], FTP [18], Gopher [2], and WAIS [10] protocols. In this way, HTTP allows basic hypermedia access to resources available from diverse applications.

1.2 Requirements

This specification uses the same words as RFC 1123 [8] for defining the significance of each particular requirement. These words are:

MUST

This word or the adjective "required" means that the item is an absolute requirement of the specification.

SHOULD

This word or the adjective "recommended" means that there may exist valid reasons in particular circumstances to ignore this item, but the full implications should be understood and the case carefully weighed before choosing a different course.

MAY

This word or the adjective "optional" means that this item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because it enhances the product, for example; another vendor may omit the same item.

An implementation is not compliant if it fails to satisfy one or more of the MUST requirements for the protocols it implements. An implementation that satisfies all the MUST and all the SHOULD requirements for its protocols is said to be "unconditionally compliant"; one that satisfies all the MUST requirements but not all the SHOULD requirements for its protocols is said to be "conditionally compliant."

1.3 Terminology

This specification uses a number of terms to refer to the roles played by participants in, and objects of, the HTTP communication.

connection

A transport layer virtual circuit established between two programs for the purpose of communication.

message

The basic unit of HTTP communication, consisting of a structured sequence of octets matching the syntax defined in section 4 and transmitted via the connection.

request

An HTTP request message, as defined in section 5.

response

An HTTP response message, as defined in section 6.

resource

A network data object or service that can be identified by a URI, as defined in section 3.2. Resources may be available in multiple representations (e.g. multiple languages, data formats, size, resolutions) or vary in other ways.

entity

The information transferred as the payload of a request or response. An entity consists of metainformation in the form of entity-header fields and content in the form of an entity-body, as described in section 7.

representation

An entity included with a response that is subject to content negotiation, as described in section 12. There may exist multiple representations associated with a particular response status.

content negotiation

The mechanism for selecting the appropriate representation when servicing a request, as described in section 12. The representation of entities in any response can be negotiated (including error responses).

variant

A resource may have one, or more than one, representation(s) associated with it at any given instant. Each of these representations is termed a 'variant.' Use of the term 'variant' does not necessarily imply that the resource is subject to content negotiation.

client

A program that establishes connections for the purpose of sending requests.

user agent

The client which initiates a request. These are often browsers, editors, spiders (web-traversing robots), or other end user tools.

server

An application program that accepts connections in order to service requests by sending back responses. Any given program may be capable of being both a client and a server; our use of these terms refers only to the role being performed by the program for a particular connection, rather than to the program's capabilities in general. Likewise, any server may act as an origin server, proxy, gateway, or tunnel, switching behavior based on the nature of each request.

origin server

The server on which a given resource resides or is to be created.

proxy

An intermediary program which acts as both a server and a client for the purpose of making requests on behalf of other clients. Requests are serviced internally or by passing them on, with possible translation, to other servers. A proxy must implement both the client and server requirements of this specification.

gateway

A server which acts as an intermediary for some other server. Unlike a proxy, a gateway receives requests as if it were the origin server for the requested resource; the requesting client may not be aware that it is communicating with a gateway.

tunnel

An intermediary program which is acting as a blind relay between two connections. Once active, a tunnel is not considered a party to the HTTP communication, though the tunnel may have been initiated by an HTTP request. The tunnel ceases to exist when both ends of the relayed connections are closed.

cache

A program's local store of response messages and the subsystem that controls its message storage, retrieval, and deletion. A cache stores cachable responses in order to reduce the response time and network bandwidth consumption on future, equivalent requests. Any client or server may include a cache, though a cache cannot be used by a server that is acting as a tunnel.

cacheable

A response is cacheable if a cache is allowed to store a copy of the response message for use in answering subsequent requests. The rules for determining the cachability of HTTP responses are defined in section 13. Even if a resource is cacheable, there may be additional constraints on whether a cache can use the cached copy for a particular request.

first-hand

A response is first-hand if it comes directly and without unnecessary delay from the origin server, perhaps via one or more proxies. A response is also first-hand if its validity has just been checked directly with the origin server.

explicit expiration time

The time at which the origin server intends that an entity should no longer be returned by a cache without further validation.

heuristic expiration time

An expiration time assigned by a cache when no explicit expiration time is available.

age

The age of a response is the time since it was sent by, or successfully validated with, the origin server.

freshness lifetime

The length of time between the generation of a response and its expiration time.

fresh

A response is fresh if its age has not yet exceeded its freshness lifetime.

stale

A response is stale if its age has passed its freshness lifetime.

semantically transparent

A cache behaves in a "semantically transparent" manner, with respect to a particular response, when its use affects neither the requesting client nor the origin server, except to improve performance. When a cache is semantically transparent, the client receives exactly the same response (except for hop-by-hop headers) that it would have received had its request been handled directly by the origin server.

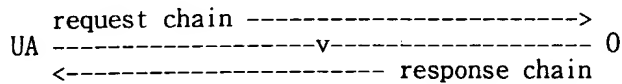
validator

A protocol element (e.g., an entity tag or a Last-Modified time) that is used to find out whether a cache entry is an equivalent copy of an entity.

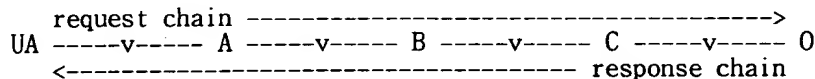
1.4 Overall Operation

The HTTP protocol is a request/response protocol. A client sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a connection with a server. The server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity metainformation, and possible entity-body content. The relationship between HTTP and MIME is described in appendix 19.4.

Most HTTP communication is initiated by a user agent and consists of a request to be applied to a resource on some origin server. In the simplest case, this may be accomplished via a single connection (v) between the user agent (UA) and the origin server (O).



A more complicated situation occurs when one or more intermediaries are present in the request/response chain. There are three common forms of intermediary: proxy, gateway, and tunnel. A proxy is a forwarding agent, receiving requests for a URI in its absolute form, rewriting all or part of the message, and forwarding the reformatted request toward the server identified by the URI. A gateway is a receiving agent, acting as a layer above some other server(s) and, if necessary, translating the requests to the underlying server's protocol. A tunnel acts as a relay point between two connections without changing the messages; tunnels are used when the communication needs to pass through an intermediary (such as a firewall) even when the intermediary cannot understand the contents of the messages.



The figure above shows three intermediaries (A, B, and C) between the user agent and origin server. A request or response message that travels the whole chain will pass through four separate connections. This distinction is important because some HTTP communication options may apply only to the connection with the nearest, non-tunnel neighbor, only to the end-points of the chain, or to all connections along the chain. Although the diagram is linear, each participant may be engaged in multiple, simultaneous communications. For example, B may be receiving requests from many clients other than A, and/or forwarding requests to servers other than C, at the same time that it is handling A's request.

Any party to the communication which is not acting as a tunnel may employ an internal cache for handling requests. The effect of a cache is that the request/response chain is shortened if one of the participants along the chain has a cached response applicable to that request. The following illustrates the resulting chain if B has a cached copy of an earlier response from O (via C) for a request which has not been cached by UA or A.

```

      request chain ----->
UA  -----v----- A -----v----- B - - - - - C - - - - - O
      <----- response chain

```

Not all responses are usefully cachable, and some requests may contain modifiers which place special requirements on cache behavior. HTTP requirements for cache behavior and cachable responses are defined in section 13.

In fact, there are a wide variety of architectures and configurations of caches and proxies currently being experimented with or deployed across the World Wide Web; these systems include national hierarchies of proxy caches to save transoceanic bandwidth, systems that broadcast or multicast cache entries, organizations that distribute subsets of cached data via CD-ROM, and so on. HTTP systems are used in corporate intranets over high-bandwidth links, and for access via PDAs with low-power radio links and intermittent connectivity. The goal of HTTP/1.1 is to support the wide diversity of configurations already deployed while introducing protocol constructs that meet the needs of those who build web applications that require high reliability and, failing that, at least reliable indications of failure.

HTTP communication usually takes place over TCP/IP connections. The default port is TCP 80, but other ports can be used. This does not preclude HTTP from being implemented on top of any other protocol on the Internet, or on other networks. HTTP only presumes a reliable transport; any protocol that provides such guarantees can be used; the mapping of the HTTP/1.1 request and response structures onto the transport data units of the protocol in question is outside the scope of this specification.

In HTTP/1.0, most implementations used a new connection for each request/response exchange. In HTTP/1.1, a connection may be used for one or more request/response exchanges, although connections may be closed for a variety of reasons (see section 8.1).

2 Notational Conventions and Generic Grammar

2.1 Augmented BNF

All of the mechanisms specified in this document are described in both prose and an augmented Backus-Naur Form (BNF) similar to that used by RFC 822 [9]. Implementers will need to be familiar with the notation in order to understand this specification. The augmented BNF includes the following constructs:

name = definition

The name of a rule is simply the name itself (without any enclosing "<" and ">") and is separated from its definition by the equal "=" character. Whitespace is only significant in that indentation of continuation lines is used to indicate a rule definition that spans more than one line. Certain basic rules are in uppercase, such as SP, LWS, HT, CRLF, DIGIT, ALPHA, etc. Angle brackets are used within definitions whenever their presence will facilitate discerning the use of rule names.

"literal"

Quotation marks surround literal text. Unless stated otherwise, the text is case-insensitive.

rule1 | rule2

Elements separated by a bar ("|") are alternatives, e.g., "yes | no" will accept yes or no.

(rule1 rule2)

Elements enclosed in parentheses are treated as a single element. Thus, "(elem (foo | bar) elem)" allows the token sequences "elem foo elem" and "elem bar elem".

***rule**

The character "*" preceding an element indicates repetition. The full form is "<n>*<m>element" indicating at least <n> and at most <m> occurrences of element. Default values are 0 and infinity so that "(element)" allows any number, including zero; "1*element" requires at least one; and "1*2element" allows one or two.

[rule]

Square brackets enclose optional elements; "[foo bar]" is equivalent to "*1(foo bar)".

N rule

Specific repetition: "<n>(element)" is equivalent to "<n>*<n>(element)"; that is, exactly <n> occurrences of (element). Thus 2DIGIT is a 2-digit number, and 3ALPHA is a string of three alphabetic characters.

#rule

A construct "#" is defined, similar to "*", for defining lists of elements. The full form is "<n>#<m>element" indicating at least <n> and at most <m> elements, each separated by one or more commas (",") and optional linear whitespace (LWS). This makes the usual form of lists very easy; a rule such as "(*LWS element *(*LWS ", " *LWS element))" can be shown as "1#element". Wherever this construct is used, null elements are allowed, but do not contribute

to the count of elements present. That is, "(element), , (element)" is permitted, but counts as only two elements. Therefore, where at least one element is required, at least one non-null element must be present. Default values are 0 and infinity so that "#element" allows any number, including zero; "1#element" requires at least one; and "1#2element" allows one or two.

; comment

A semi-colon, set off some distance to the right of rule text, starts a comment that continues to the end of line. This is a simple way of including useful notes in parallel with the specifications.

implied *LWS

The grammar described by this specification is word-based. Except where noted otherwise, linear whitespace (LWS) can be included between any two adjacent words (token or quoted-string), and between adjacent tokens and delimiters (tspecials), without changing the interpretation of a field. At least one delimiter (tspecials) must exist between any two tokens, since they would otherwise be interpreted as a single token.

2.2 Basic Rules

The following rules are used throughout this specification to describe basic parsing constructs. The US-ASCII coded character set is defined by ANSI X3.4-1986 [21].

OCTET	= <any 8-bit sequence of data>
CHAR	= <any US-ASCII character (octets 0 - 127)>
UPALPHA	= <any US-ASCII uppercase letter "A".."Z">
LOALPHA	= <any US-ASCII lowercase letter "a".."z">
ALPHA	= UPALPHA LOALPHA
DIGIT	= <any US-ASCII digit "0".."9">
CTL	= <any US-ASCII control character (octets 0 - 31) and DEL (127)>
CR	= <US-ASCII CR, carriage return (13)>
LF	= <US-ASCII LF, linefeed (10)>
SP	= <US-ASCII SP, space (32)>
HT	= <US-ASCII HT, horizontal-tab (9)>
<">	= <US-ASCII double-quote mark (34)>

HTTP/1.1 defines the sequence CR LF as the end-of-line marker for all protocol elements except the entity-body (see appendix 19.3 for tolerant applications). The end-of-line marker within an entity-body is defined by its associated media type, as described in section 3.7.

CRLF = CR LF

HTTP/1.1 headers can be folded onto multiple lines if the continuation line begins with a space or horizontal tab. All linear white space, including folding, has the same semantics as SP.

LWS = [CRLF] 1*(SP | HT)

The TEXT rule is only used for descriptive field contents and values that are not intended to be interpreted by the message parser. Words of *TEXT may contain characters from character sets other than ISO 8859-1 [22] only when encoded according to the rules of RFC 1522 [14].

TEXT = <any OCTET except CTLs,
but including LWS>

Hexadecimal numeric characters are used in several protocol elements.

HEX = "A" | "B" | "C" | "D" | "E" | "F"
| "a" | "b" | "c" | "d" | "e" | "f" | DIGIT

Many HTTP/1.1 header field values consist of words separated by LWS or special characters. These special characters MUST be in a quoted string to be used within a parameter value.

token = 1*<any CHAR except CTLs or tspecials>

tspecials = "(" | ")" | "<" | ">" | "@"
| "\"" | "." | ":" | ";" | "<" | ">"
| "/" | "[" | "]" | "?" | "="
| "{" | "}" | SP | HT

Comments can be included in some HTTP header fields by surrounding the comment text with parentheses. Comments are only allowed in fields containing "comment" as part of their field value definition. In all other fields, parentheses are considered part of the field value.

comment = "(" *(ctext | comment) ")"
ctext = <any TEXT excluding "(" and ">">

A string of text is parsed as a single word if it is quoted using double-quote marks.

quoted-string = (<"> *(qdtype) <">)

qdtype = <any TEXT except <">>

The backslash character ("\") may be used as a single-character quoting mechanism only within quoted-string and comment constructs.

quoted-pair = "\" CHAR

3 Protocol Parameters

3.1 HTTP Version

HTTP uses a "<major>.<minor>" numbering scheme to indicate versions of the protocol. The protocol versioning policy is intended to allow the sender to indicate the format of a message and its capacity for understanding further HTTP communication, rather than the features obtained via that communication. No change is made to the version number for the addition of message components which do not affect communication behavior or which only add to extensible field values. The <minor> number is incremented when the changes made to the protocol add features which do not change the general message parsing algorithm, but which may add to the message semantics and imply additional capabilities of the sender. The <major> number is incremented when the format of a message within the protocol is changed.

The version of an HTTP message is indicated by an HTTP-Version field in the first line of the message.

HTTP-Version = "HTTP" "/" 1*DIGIT "." 1*DIGIT

Note that the major and minor numbers MUST be treated as separate integers and that each may be incremented higher than a single digit. Thus, HTTP/2.4 is a lower version than HTTP/2.13, which in turn is lower than HTTP/12.3. Leading zeros MUST be ignored by recipients and MUST NOT be sent.

Applications sending Request or Response messages, as defined by this specification, MUST include an HTTP-Version of "HTTP/1.1". Use of this version number indicates that the sending application is at least conditionally compliant with this specification.

The HTTP version of an application is the highest HTTP version for which the application is at least conditionally compliant.

Proxy and gateway applications must be careful when forwarding messages in protocol versions different from that of the application. Since the protocol version indicates the protocol capability of the sender, a proxy/gateway **MUST** never send a message with a version indicator which is greater than its actual version; if a higher version request is received, the proxy/gateway **MUST** either downgrade the request version, respond with an error, or switch to tunnel behavior. Requests with a version lower than that of the proxy/gateway's version **MAY** be upgraded before being forwarded; the proxy/gateway's response to that request **MUST** be in the same major version as the request.

Note: Converting between versions of HTTP may involve modification of header fields required or forbidden by the versions involved.

3.2 Uniform Resource Identifiers

URIs have been known by many names: WWW addresses, Universal Document Identifiers, Universal Resource Identifiers, and finally the combination of Uniform Resource Locators (URL) and Names (URN). As far as HTTP is concerned, Uniform Resource Identifiers are simply formatted strings which identify--via name, location, or any other characteristic--a resource.

3.2.1 General Syntax

URIs in HTTP can be represented in absolute form or relative to some known base URI, depending upon the context of their use. The two forms are differentiated by the fact that absolute URIs always begin with a scheme name followed by a colon.

```

URI           = ( absoluteURI | relativeURI ) [ "#" fragment ]
absoluteURI   = scheme ":" *( uchar | reserved )
relativeURI   = net_path | abs_path | rel_path
net_path      = "//" net_loc [ abs_path ]
abs_path      = "/" rel_path
rel_path      = [ path ] [ ";" params ] [ "?" query ]

path          = fsegment *( "/" segment )
fsegment      = 1*pchar
segment       = *pchar

params        = param *( ";" param )
param         = *( pchar | "/" )

```

```

scheme      = 1*( ALPHA | DIGIT | "+" | "-" | "." )
net_loc     = *( pchar | ";" | "?" )

query       = *( uchar | reserved )
fragment    = *( uchar | reserved )

pchar       = uchar | ":" | "@" | "&" | "=" | "+"
uchar       = unreserved | escape
unreserved  = ALPHA | DIGIT | safe | extra | national

escape      = "%" HEX HEX
reserved    = ";" | "/" | "?" | ":" | "@" | "&" | "=" | "+"
extra       = "!" | "*" | "'" | "(" | ")" | ","
safe        = "$" | "_" | "." | "-"
unsafe      = CTL | SP | "<" | ">" | "#" | "%" | "<" | ">"
national    = <any OCTET excluding ALPHA, DIGIT,
              reserved, extra, safe, and unsafe>

```

For definitive information on URL syntax and semantics, see RFC 1738 [4] and RFC 1808 [11]. The BNF above includes national characters not allowed in valid URLs as specified by RFC 1738, since HTTP servers are not restricted in the set of unreserved characters allowed to represent the `rel_path` part of addresses, and HTTP proxies may receive requests for URIs not defined by RFC 1738.

The HTTP protocol does not place any a priori limit on the length of a URI. Servers **MUST** be able to handle the URI of any resource they serve, and **SHOULD** be able to handle URIs of unbounded length if they provide GET-based forms that could generate such URIs. A server **SHOULD** return 414 (Request-URI Too Long) status if a URI is longer than the server can handle (see section 10.4.15).

Note: Servers should be cautious about depending on URI lengths above 255 bytes, because some older client or proxy implementations may not properly support these lengths.

3.2.2 http URL

The "http" scheme is used to locate network resources via the HTTP protocol. This section defines the scheme-specific syntax and semantics for http URLs.

http_URL = "http:" "/" host [":" port] [abs_path]
host = <A legal Internet host domain name
or IP address (in dotted-decimal form),
as defined by Section 2.1 of RFC 1123>
port = *DIGIT

If the port is empty or not given, port 80 is assumed. The semantics are that the identified resource is located at the server listening for TCP connections on that port of that host, and the Request-URI for the resource is abs_path. The use of IP addresses in URL's SHOULD be avoided whenever possible (see RFC 1900 [24]). If the abs_path is not present in the URL, it MUST be given as "/" when used as a Request-URI for a resource (section 5.1.2).

3.2.3 URI Comparison

When comparing two URIs to decide if they match or not, a client SHOULD use a case-sensitive octet-by-octet comparison of the entire URIs, with these exceptions:

- o A port that is empty or not given is equivalent to the default port for that URI;
- o Comparisons of host names MUST be case-insensitive;
- o Comparisons of scheme names MUST be case-insensitive;
- o An empty abs_path is equivalent to an abs_path of "/".

Characters other than those in the "reserved" and "unsafe" sets (see section 3.2) are equivalent to their "%" HEX HEX" encodings.

For example, the following three URIs are equivalent:

```
http://abc.com:80/~smith/home.html  
http://ABC.com/%7Esmith/home.html  
http://ABC.com:/%7esmith/home.html
```

3.3 Date/Time Formats

3.3.1 Full Date

HTTP applications have historically allowed three different formats for the representation of date/time stamps:

```
Sun, 06 Nov 1994 08:49:37 GMT ; RFC 822, updated by RFC 1123
Sunday, 06-Nov-94 08:49:37 GMT ; RFC 850, obsoleted by RFC 1036
Sun Nov  6 08:49:37 1994      ; ANSI C's asctime() format
```

The first format is preferred as an Internet standard and represents a fixed-length subset of that defined by RFC 1123 (an update to RFC 822). The second format is in common use, but is based on the obsolete RFC 850 [12] date format and lacks a four-digit year. HTTP/1.1 clients and servers that parse the date value MUST accept all three formats (for compatibility with HTTP/1.0), though they MUST only generate the RFC 1123 format for representing HTTP-date values in header fields.

Note: Recipients of date values are encouraged to be robust in accepting date values that may have been sent by non-HTTP applications, as is sometimes the case when retrieving or posting messages via proxies/gateways to SMTP or NNTP.

All HTTP date/time stamps MUST be represented in Greenwich Mean Time (GMT), without exception. This is indicated in the first two formats by the inclusion of "GMT" as the three-letter abbreviation for time zone, and MUST be assumed when reading the asctime format.

```
HTTP-date    = rfc1123-date | rfc850-date | asctime-date
```

```
rfc1123-date = wkday "," SP date1 SP time SP "GMT"
rfc850-date  = weekday "," SP date2 SP time SP "GMT"
asctime-date = wkday SP date3 SP time SP 4DIGIT
```

```
date1        = 2DIGIT SP month SP 4DIGIT
               ; day month year (e.g., 02 Jun 1982)
date2        = 2DIGIT "-" month "-" 2DIGIT
               ; day-month-year (e.g., 02-Jun-82)
date3        = month SP ( 2DIGIT | ( SP 1DIGIT ) )
               ; month day (e.g., Jun  2)
```

```
time         = 2DIGIT ":" 2DIGIT ":" 2DIGIT
               ; 00:00:00 - 23:59:59
```

```
wkday        = "Mon" | "Tue" | "Wed"
               | "Thu" | "Fri" | "Sat" | "Sun"
```

```
weekday = "Monday" | "Tuesday" | "Wednesday"  
         | "Thursday" | "Friday" | "Saturday" | "Sunday"  
  
month   = "Jan" | "Feb" | "Mar" | "Apr"  
         | "May" | "Jun" | "Jul" | "Aug"  
         | "Sep" | "Oct" | "Nov" | "Dec"
```

Note: HTTP requirements for the date/time stamp format apply only to their usage within the protocol stream. Clients and servers are not required to use these formats for user presentation, request logging, etc.

3.3.2 Delta Seconds

Some HTTP header fields allow a time value to be specified as an integer number of seconds, represented in decimal, after the time that the message was received.

delta-seconds = 1*DIGIT

3.4 Character Sets

HTTP uses the same definition of the term "character set" as that described for MIME:

The term "character set" is used in this document to refer to a method used with one or more tables to convert a sequence of octets into a sequence of characters. Note that unconditional conversion in the other direction is not required, in that not all characters may be available in a given character set and a character set may provide more than one sequence of octets to represent a particular character. This definition is intended to allow various kinds of character encodings, from simple single-table mappings such as US-ASCII to complex table switching methods such as those that use ISO 2022's techniques. However, the definition associated with a MIME character set name **MUST** fully specify the mapping to be performed from octets to characters. In particular, use of external profiling information to determine the exact mapping is not permitted.

Note: This use of the term "character set" is more commonly referred to as a "character encoding." However, since HTTP and MIME share the same registry, it is important that the terminology also be shared.

HTTP character sets are identified by case-insensitive tokens. The complete set of tokens is defined by the IANA Character Set registry [19].

charset = token

Although HTTP allows an arbitrary token to be used as a charset value, any token that has a predefined value within the IANA Character Set registry MUST represent the character set defined by that registry. Applications SHOULD limit their use of character sets to those defined by the IANA registry.

3.5 Content Codings

Content coding values indicate an encoding transformation that has been or can be applied to an entity. Content codings are primarily used to allow a document to be compressed or otherwise usefully transformed without losing the identity of its underlying media type and without loss of information. Frequently, the entity is stored in coded form, transmitted directly, and only decoded by the recipient.

content-coding = token

All content-coding values are case-insensitive. HTTP/1.1 uses content-coding values in the Accept-Encoding (section 14.3) and Content-Encoding (section 14.12) header fields. Although the value describes the content-coding, what is more important is that it indicates what decoding mechanism will be required to remove the encoding.

The Internet Assigned Numbers Authority (IANA) acts as a registry for content-coding value tokens. Initially, the registry contains the following tokens:

gzip An encoding format produced by the file compression program "gzip" (GNU zip) as described in RFC 1952 [25]. This format is a Lempel-Ziv coding (LZ77) with a 32 bit CRC.

compress

The encoding format produced by the common UNIX file compression program "compress". This format is an adaptive Lempel-Ziv-Welch coding (LZW).

Note: Use of program names for the identification of encoding formats is not desirable and should be discouraged for future encodings. Their use here is representative of historical practice, not good design. For compatibility with previous implementations of HTTP, applications should consider "x-gzip" and "x-compress" to be equivalent to "gzip" and "compress" respectively.

deflate The "zlib" format defined in RFC 1950[31] in combination with the "deflate" compression mechanism described in RFC 1951[29].

New content-coding value tokens should be registered; to allow interoperability between clients and servers, specifications of the content coding algorithms needed to implement a new value should be publicly available and adequate for independent implementation, and conform to the purpose of content coding defined in this section.

3.6 Transfer Codings

Transfer coding values are used to indicate an encoding transformation that has been, can be, or may need to be applied to an entity-body in order to ensure "safe transport" through the network. This differs from a content coding in that the transfer coding is a property of the message, not of the original entity.

transfer-coding = "chunked" | transfer-extension
transfer-extension = token

All transfer-coding values are case-insensitive. HTTP/1.1 uses transfer coding values in the Transfer-Encoding header field (section 14.40).

Transfer codings are analogous to the Content-Transfer-Encoding values of MIME, which were designed to enable safe transport of binary data over a 7-bit transport service. However, safe transport has a different focus for an 8bit-clean transfer protocol. In HTTP, the only unsafe characteristic of message-bodies is the difficulty in determining the exact body length (section 7.2.2), or the desire to encrypt data over a shared transport.

The chunked encoding modifies the body of a message in order to transfer it as a series of chunks, each with its own size indicator, followed by an optional footer containing entity-header fields. This allows dynamically-produced content to be transferred along with the information necessary for the recipient to verify that it has received the full message.

```

Chunked-Body = *chunk
               "0" CRLF
               footer
               CRLF

chunk         = chunk-size [ chunk-ext ] CRLF
               chunk-data CRLF

hex-no-zero   = <HEX excluding "0">

chunk-size    = hex-no-zero *HEX
chunk-ext     = *( ";" chunk-ext-name [ "=" chunk-ext-value ] )
chunk-ext-name = token
chunk-ext-val  = token | quoted-string
chunk-data    = chunk-size(OCTET)

footer        = *entity-header

```

The chunked encoding is ended by a zero-sized chunk followed by the footer, which is terminated by an empty line. The purpose of the footer is to provide an efficient way to supply information about an entity that is generated dynamically; applications **MUST NOT** send header fields in the footer which are not explicitly defined as being appropriate for the footer, such as Content-MD5 or future extensions to HTTP for digital signatures or other facilities.

An example process for decoding a Chunked-Body is presented in appendix 19.4.6.

All HTTP/1.1 applications **MUST** be able to receive and decode the "chunked" transfer coding, and **MUST** ignore transfer coding extensions they do not understand. A server which receives an entity-body with a transfer-coding it does not understand **SHOULD** return 501 (Unimplemented), and close the connection. A server **MUST NOT** send transfer-codings to an HTTP/1.0 client.

3.7 Media Types

HTTP uses Internet Media Types in the Content-Type (section 14.18) and Accept (section 14.1) header fields in order to provide open and extensible data typing and type negotiation.

```

media-type    = type "/" subtype *( ";" parameter )
type          = token
subtype       = token

```

Parameters may follow the type/subtype in the form of attribute/value pairs.

parameter	= attribute "=" value
attribute	= token
value	= token quoted-string

The type, subtype, and parameter attribute names are case-insensitive. Parameter values may or may not be case-sensitive, depending on the semantics of the parameter name. Linear white space (LWS) MUST NOT be used between the type and subtype, nor between an attribute and its value. User agents that recognize the media-type MUST process (or arrange to be processed by any external applications used to process that type/subtype by the user agent) the parameters for that MIME type as described by that type/subtype definition to the and inform the user of any problems discovered.

Note: some older HTTP applications do not recognize media type parameters. When sending data to older HTTP applications, implementations should only use media type parameters when they are required by that type/subtype definition.

Media-type values are registered with the Internet Assigned Number Authority (IANA). The media type registration process is outlined in RFC 2048 [17]. Use of non-registered media types is discouraged.

3.7.1 Canonicalization and Text Defaults

Internet media types are registered with a canonical form. In general, an entity-body transferred via HTTP messages MUST be represented in the appropriate canonical form prior to its transmission; the exception is "text" types, as defined in the next paragraph.

When in canonical form, media subtypes of the "text" type use CRLF as the text line break. HTTP relaxes this requirement and allows the transport of text media with plain CR or LF alone representing a line break when it is done consistently for an entire entity-body. HTTP applications MUST accept CRLF, bare CR, and bare LF as being representative of a line break in text media received via HTTP. In addition, if the text is represented in a character set that does not use octets 13 and 10 for CR and LF respectively, as is the case for some multi-byte character sets, HTTP allows the use of whatever octet sequences are defined by that character set to represent the equivalent of CR and LF for line breaks. This flexibility regarding line breaks applies only to text media in the entity-body; a bare CR or LF MUST NOT be substituted for CRLF within any of the HTTP control structures (such as header fields and multipart boundaries).

If an entity-body is encoded with a Content-Encoding, the underlying data MUST be in a form defined above prior to being encoded.

The "charset" parameter is used with some media types to define the character set (section 3.4) of the data. When no explicit charset parameter is provided by the sender, media subtypes of the "text" type are defined to have a default charset value of "ISO-8859-1" when received via HTTP. Data in character sets other than "ISO-8859-1" or its subsets **MUST** be labeled with an appropriate charset value.

Some HTTP/1.0 software has interpreted a Content-Type header without charset parameter incorrectly to mean "recipient should guess." Senders wishing to defeat this behavior **MAY** include a charset parameter even when the charset is ISO-8859-1 and **SHOULD** do so when it is known that it will not confuse the recipient.

Unfortunately, some older HTTP/1.0 clients did not deal properly with an explicit charset parameter. HTTP/1.1 recipients **MUST** respect the charset label provided by the sender; and those user agents that have a provision to "guess" a charset **MUST** use the charset from the content-type field if they support that charset, rather than the recipient's preference, when initially displaying a document.

3.7.2 Multipart Types

MIME provides for a number of "multipart" types -- encapsulations of one or more entities within a single message-body. All multipart types share a common syntax, as defined in MIME [7], and **MUST** include a boundary parameter as part of the media type value. The message body is itself a protocol element and **MUST** therefore use only CRLF to represent line breaks between body-parts. Unlike in MIME, the epilogue of any multipart message **MUST** be empty; HTTP applications **MUST NOT** transmit the epilogue (even if the original multipart contains an epilogue).

In HTTP, multipart body-parts **MAY** contain header fields which are significant to the meaning of that part. A Content-Location header field (section 14.15) **SHOULD** be included in the body-part of each enclosed entity that can be identified by a URL.

In general, an HTTP user agent **SHOULD** follow the same or similar behavior as a MIME user agent would upon receipt of a multipart type. If an application receives an unrecognized multipart subtype, the application **MUST** treat it as being equivalent to "multipart/mixed".

Note: The "multipart/form-data" type has been specifically defined for carrying form data suitable for processing via the POST request method, as described in RFC 1867 [15].

3.8 Product Tokens

Product tokens are used to allow communicating applications to identify themselves by software name and version. Most fields using product tokens also allow sub-products which form a significant part of the application to be listed, separated by whitespace. By convention, the products are listed in order of their significance for identifying the application.

product = token ["/" product-version]
product-version = token

Examples:

User-Agent: CERN-LineMode/2.15 libwww/2.17b3
Server: Apache/0.8.4

Product tokens should be short and to the point -- use of them for advertising or other non-essential information is explicitly forbidden. Although any token character may appear in a product-version, this token **SHOULD** only be used for a version identifier (i.e., successive versions of the same product **SHOULD** only differ in the product-version portion of the product value).

3.9 Quality Values

HTTP content negotiation (section 12) uses short "floating point" numbers to indicate the relative importance ("weight") of various negotiable parameters. A weight is normalized to a real number in the range 0 through 1, where 0 is the minimum and 1 the maximum value. HTTP/1.1 applications **MUST NOT** generate more than three digits after the decimal point. User configuration of these values **SHOULD** also be limited in this fashion.

qvalue = ("0" ["." 0*3DIGIT])
| ("1" ["." 0*3("0")])

"Quality values" is a misnomer, since these values merely represent relative degradation in desired quality.

3.10 Language Tags

A language tag identifies a natural language spoken, written, or otherwise conveyed by human beings for communication of information to other human beings. Computer languages are explicitly excluded. HTTP uses language tags within the Accept-Language and Content-Language fields.

The syntax and registry of HTTP language tags is the same as that defined by RFC 1766 [1]. In summary, a language tag is composed of 1 or more parts: A primary language tag and a possibly empty series of subtags:

language-tag = primary-tag *("-" subtag)

primary-tag = 1*8ALPHA

subtag = 1*8ALPHA

Whitespace is not allowed within the tag and all tags are case-insensitive. The name space of language tags is administered by the IANA. Example tags include:

en, en-US, en-cockney, i-cherokee, x-pig-latin

where any two-letter primary-tag is an ISO 639 language abbreviation and any two-letter initial subtag is an ISO 3166 country code. (The last three tags above are not registered tags; all but the last are examples of tags which could be registered in future.)

3.11 Entity Tags

Entity tags are used for comparing two or more entities from the same requested resource. HTTP/1.1 uses entity tags in the ETag (section 14.20), If-Match (section 14.25), If-None-Match (section 14.26), and If-Range (section 14.27) header fields. The definition of how they are used and compared as cache validators is in section 13.3.3. An entity tag consists of an opaque quoted string, possibly prefixed by a weakness indicator.

entity-tag = [weak] opaque-tag

weak = "W/"

opaque-tag = quoted-string

A "strong entity tag" may be shared by two entities of a resource only if they are equivalent by octet equality.

A "weak entity tag," indicated by the "W/" prefix, may be shared by two entities of a resource only if the entities are equivalent and could be substituted for each other with no significant change in semantics. A weak entity tag can only be used for weak comparison.

An entity tag **MUST** be unique across all versions of all entities associated with a particular resource. A given entity tag value may be used for entities obtained by requests on different URIs without implying anything about the equivalence of those entities.

3.12 Range Units

HTTP/1.1 allows a client to request that only part (a range of) the response entity be included within the response. HTTP/1.1 uses range units in the Range (section 14.36) and Content-Range (section 14.17) header fields. An entity may be broken down into subranges according to various structural units.

range-unit = bytes-unit | other-range-unit

bytes-unit = "bytes"

other-range-unit = token

The only range unit defined by HTTP/1.1 is "bytes". HTTP/1.1 implementations may ignore ranges specified using other units. HTTP/1.1 has been designed to allow implementations of applications that do not depend on knowledge of ranges.

4 HTTP Message

4.1 Message Types

HTTP messages consist of requests from client to server and responses from server to client.

HTTP-message = Request | Response ; HTTP/1.1 messages

Request (section 5) and Response (section 6) messages use the generic message format of RFC 822 [9] for transferring entities (the payload of the message). Both types of message consist of a start-line, one or more header fields (also known as "headers"), an empty line (i.e., a line with nothing preceding the CRLF) indicating the end of the header fields, and an optional message-body.

generic-message = start-line
 *message-header
 CRLF
 [message-body]

start-line = Request-Line | Status-Line

In the interest of robustness, servers SHOULD ignore any empty line(s) received where a Request-Line is expected. In other words, if the server is reading the protocol stream at the beginning of a message and receives a CRLF first, it should ignore the CRLF.

Note: certain buggy HTTP/1.0 client implementations generate an extra CRLF's after a POST request. To restate what is explicitly forbidden by the BNF, an HTTP/1.1 client must not preface or follow a request with an extra CRLF.

4.2 Message Headers

HTTP header fields, which include general-header (section 4.5), request-header (section 5.3), response-header (section 6.2), and entity-header (section 7.1) fields, follow the same generic format as that given in Section 3.1 of RFC 822 [9]. Each header field consists of a name followed by a colon (":") and the field value. Field names are case-insensitive. The field value may be preceded by any amount of LWS, though a single SP is preferred. Header fields can be extended over multiple lines by preceding each extra line with at least one SP or HT. Applications SHOULD follow "common form" when generating HTTP constructs, since there might exist some implementations that fail to accept anything beyond the common forms.

message-header = field-name ":" [field-value] CRLF

field-name = token

field-value = *(field-content | LWS)

field-content = <the OCTETs making up the field-value
and consisting of either *TEXT or combinations
of token, tspecials, and quoted-string>

The order in which header fields with differing field names are received is not significant. However, it is "good practice" to send general-header fields first, followed by request-header or response-header fields, and ending with the entity-header fields.

Multiple message-header fields with the same field-name may be present in a message if and only if the entire field-value for that header field is defined as a comma-separated list [i.e., #(values)]. It MUST be possible to combine the multiple header fields into one "field-name: field-value" pair, without changing the semantics of the message, by appending each subsequent field-value to the first, each separated by a comma. The order in which header fields with the same field-name are received is therefore significant to the interpretation of the combined field value, and thus a proxy MUST NOT change the order of these field values when a message is forwarded.

4.3 Message Body

The message-body (if any) of an HTTP message is used to carry the entity-body associated with the request or response. The message-body differs from the entity-body only when a transfer coding has been applied, as indicated by the Transfer-Encoding header field (section 14.40).

message-body = entity-body
| <entity-body encoded as per Transfer-Encoding>

Transfer-Encoding MUST be used to indicate any transfer codings applied by an application to ensure safe and proper transfer of the message. Transfer-Encoding is a property of the message, not of the entity, and thus can be added or removed by any application along the request/response chain.

The rules for when a message-body is allowed in a message differ for requests and responses.

The presence of a message-body in a request is signaled by the inclusion of a Content-Length or Transfer-Encoding header field in the request's message-headers. A message-body MAY be included in a request only when the request method (section 5.1.1) allows an entity-body.

For response messages, whether or not a message-body is included with a message is dependent on both the request method and the response status code (section 6.1.1). All responses to the HEAD request method MUST NOT include a message-body, even though the presence of entity-header fields might lead one to believe they do. All 1xx (informational), 204 (no content), and 304 (not modified) responses MUST NOT include a message-body. All other responses do include a message-body, although it may be of zero length.

4.4 Message Length

When a message-body is included with a message, the length of that body is determined by one of the following (in order of precedence):

1. Any response message which MUST NOT include a message-body (such as the 1xx, 204, and 304 responses and any response to a HEAD request) is always terminated by the first empty line after the header fields, regardless of the entity-header fields present in the message.
2. If a Transfer-Encoding header field (section 14.40) is present and indicates that the "chunked" transfer coding has been applied, then

the length is defined by the chunked encoding (section 3.6).

3. If a Content-Length header field (section 14.14) is present, its value in bytes represents the length of the message-body.
4. If the message uses the media type "multipart/byteranges", which is self-delimiting, then that defines the length. This media type **MUST** NOT be used unless the sender knows that the recipient can parse it; the presence in a request of a Range header with multiple byte-range specifiers implies that the client can parse multipart/byteranges responses.
5. By the server closing the connection. (Closing the connection cannot be used to indicate the end of a request body, since that would leave no possibility for the server to send back a response.)

For compatibility with HTTP/1.0 applications, HTTP/1.1 requests containing a message-body **MUST** include a valid Content-Length header field unless the server is known to be HTTP/1.1 compliant. If a request contains a message-body and a Content-Length is not given, the server **SHOULD** respond with 400 (bad request) if it cannot determine the length of the message, or with 411 (length required) if it wishes to insist on receiving a valid Content-Length.

All HTTP/1.1 applications that receive entities **MUST** accept the "chunked" transfer coding (section 3.6), thus allowing this mechanism to be used for messages when the message length cannot be determined in advance.

Messages **MUST NOT** include both a Content-Length header field and the "chunked" transfer coding. If both are received, the Content-Length **MUST** be ignored.

When a Content-Length is given in a message where a message-body is allowed, its field value **MUST** exactly match the number of OCTETs in the message-body. HTTP/1.1 user agents **MUST** notify the user when an invalid length is received and detected.

4.5 General Header Fields

There are a few header fields which have general applicability for both request and response messages, but which do not apply to the entity being transferred. These header fields apply only to the message being transmitted.

general-header	=	Cache-Control	:	Section 14.9
		Connection	:	Section 14.10
		Date	:	Section 14.19
		Pragma	:	Section 14.32
		Transfer-Encoding	:	Section 14.40
		Upgrade	:	Section 14.41
		Via	:	Section 14.44

General-header field names can be extended reliably only in combination with a change in the protocol version. However, new or experimental header fields may be given the semantics of general header fields if all parties in the communication recognize them to be general-header fields. Unrecognized header fields are treated as entity-header fields.

5 Request

A request message from a client to a server includes, within the first line of that message, the method to be applied to the resource, the identifier of the resource, and the protocol version in use.

Request	=	Request-Line	:	Section 5.1
	*	(general-header	:	Section 4.5
		request-header	:	Section 5.3
		entity-header)	:	Section 7.1
		CRLF		
		[message-body]	:	Section 7.2

5.1 Request-Line

The Request-Line begins with a method token, followed by the Request-URI and the protocol version, and ending with CRLF. The elements are separated by SP characters. No CR or LF are allowed except in the final CRLF sequence.

Request-Line = Method SP Request-URI SP HTTP-Version CRLF

5.1.1 Method

The Method token indicates the method to be performed on the resource identified by the Request-URI. The method is case-sensitive.

Method	= "OPTIONS"	; Section 9.2
	"GET"	; Section 9.3
	"HEAD"	; Section 9.4
	"POST"	; Section 9.5
	"PUT"	; Section 9.6
	"DELETE"	; Section 9.7
	"TRACE"	; Section 9.8
	extension-method	

extension-method = token

The list of methods allowed by a resource can be specified in an Allow header field (section 14.7). The return code of the response always notifies the client whether a method is currently allowed on a resource, since the set of allowed methods can change dynamically. Servers SHOULD return the status code 405 (Method Not Allowed) if the method is known by the server but not allowed for the requested resource, and 501 (Not Implemented) if the method is unrecognized or not implemented by the server. The list of methods known by a server can be listed in a Public response-header field (section 14.35).

The methods GET and HEAD MUST be supported by all general-purpose servers. All other methods are optional; however, if the above methods are implemented, they MUST be implemented with the same semantics as those specified in section 9.

5.1.2 Request-URI

The Request-URI is a Uniform Resource Identifier (section 3.2) and identifies the resource upon which to apply the request.

Request-URI = "*" | absoluteURI | abs_path

The three options for Request-URI are dependent on the nature of the request. The asterisk "*" means that the request does not apply to a particular resource, but to the server itself, and is only allowed when the method used does not necessarily apply to a resource. One example would be

OPTIONS * HTTP/1.1

The absoluteURI form is required when the request is being made to a proxy. The proxy is requested to forward the request or service it

from a valid cache, and return the response. Note that the proxy MAY forward the request on to another proxy or directly to the server specified by the absoluteURI. In order to avoid request loops, a proxy MUST be able to recognize all of its server names, including any aliases, local variations, and the numeric IP address. An example Request-Line would be:

```
GET http://www.w3.org/pub/WWW/TheProject.html HTTP/1.1
```

To allow for transition to absoluteURIs in all requests in future versions of HTTP, all HTTP/1.1 servers MUST accept the absoluteURI form in requests, even though HTTP/1.1 clients will only generate them in requests to proxies.

The most common form of Request-URI is that used to identify a resource on an origin server or gateway. In this case the absolute path of the URI MUST be transmitted (see section 3.2.1, *abs_path*) as the Request-URI, and the network location of the URI (*net_loc*) MUST be transmitted in a Host header field. For example, a client wishing to retrieve the resource above directly from the origin server would create a TCP connection to port 80 of the host "www.w3.org" and send the lines:

```
GET /pub/WWW/TheProject.html HTTP/1.1
Host: www.w3.org
```

followed by the remainder of the Request. Note that the absolute path cannot be empty; if none is present in the original URI, it MUST be given as "/" (the server root).

If a proxy receives a request without any path in the Request-URI and the method specified is capable of supporting the asterisk form of request, then the last proxy on the request chain MUST forward the request with "*" as the final Request-URI. For example, the request

```
OPTIONS http://www.ics.uci.edu:8001 HTTP/1.1
```

would be forwarded by the proxy as

```
OPTIONS * HTTP/1.1
Host: www.ics.uci.edu:8001
```

after connecting to port 8001 of host "www.ics.uci.edu".

The Request-URI is transmitted in the format specified in section 3.2.1. The origin server MUST decode the Request-URI in order to properly interpret the request. Servers SHOULD respond to invalid Request-URIs with an appropriate status code.

In requests that they forward, proxies **MUST NOT** rewrite the "abs_path" part of a Request-URI in any way except as noted above to replace a null abs_path with "*", no matter what the proxy does in its internal implementation.

Note: The "no rewrite" rule prevents the proxy from changing the meaning of the request when the origin server is improperly using a non-reserved URL character for a reserved purpose. Implementers should be aware that some pre-HTTP/1.1 proxies have been known to rewrite the Request-URI.

5.2 The Resource Identified by a Request

HTTP/1.1 origin servers **SHOULD** be aware that the exact resource identified by an Internet request is determined by examining both the Request-URI and the Host header field.

An origin server that does not allow resources to differ by the requested host **MAY** ignore the Host header field value. (But see section 19.5.1 for other requirements on Host support in HTTP/1.1.)

An origin server that does differentiate resources based on the host requested (sometimes referred to as virtual hosts or vanity hostnames) **MUST** use the following rules for determining the requested resource on an HTTP/1.1 request:

1. If Request-URI is an absoluteURI, the host is part of the Request-URI. Any Host header field value in the request **MUST** be ignored.
2. If the Request-URI is not an absoluteURI, and the request includes a Host header field, the host is determined by the Host header field value.
3. If the host as determined by rule 1 or 2 is not a valid host on the server, the response **MUST** be a 400 (Bad Request) error message.

Recipients of an HTTP/1.0 request that lacks a Host header field **MAY** attempt to use heuristics (e.g., examination of the URI path for something unique to a particular host) in order to determine what exact resource is being requested.

5.3 Request Header Fields

The request-header fields allow the client to pass additional information about the request, and about the client itself, to the server. These fields act as request modifiers, with semantics

equivalent to the parameters on a programming language method invocation.

```

request-header = Accept           ; Section 14.1
                  | Accept-Charset ; Section 14.2
                  | Accept-Encoding ; Section 14.3
                  | Accept-Language ; Section 14.4
                  | Authorization  ; Section 14.8
                  | From           ; Section 14.22
                  | Host           ; Section 14.23
                  | If-Modified-Since ; Section 14.24
                  | If-Match       ; Section 14.25
                  | If-None-Match  ; Section 14.26
                  | If-Range       ; Section 14.27
                  | If-Unmodified-Since ; Section 14.28
                  | Max-Forwards   ; Section 14.31
                  | Proxy-Authorization ; Section 14.34
                  | Range          ; Section 14.36
                  | Referer        ; Section 14.37
                  | User-Agent     ; Section 14.42

```

Request-header field names can be extended reliably only in combination with a change in the protocol version. However, new or experimental header fields MAY be given the semantics of request-header fields if all parties in the communication recognize them to be request-header fields. Unrecognized header fields are treated as entity-header fields.

6 Response

After receiving and interpreting a request message, a server responds with an HTTP response message.

```

Response = Status-Line           ; Section 6.1
          *( general-header       ; Section 4.5
            | response-header     ; Section 6.2
            | entity-header )     ; Section 7.1
          CRLF
          [ message-body ]       ; Section 7.2

```

6.1 Status-Line

The first line of a Response message is the Status-Line, consisting of the protocol version followed by a numeric status code and its associated textual phrase, with each element separated by SP characters. No CR or LF is allowed except in the final CRLF sequence.

Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF

6.1.1 Status Code and Reason Phrase

The Status-Code element is a 3-digit integer result code of the attempt to understand and satisfy the request. These codes are fully defined in section 10. The Reason-Phrase is intended to give a short textual description of the Status-Code. The Status-Code is intended for use by automata and the Reason-Phrase is intended for the human user. The client is not required to examine or display the Reason-Phrase.

The first digit of the Status-Code defines the class of response. The last two digits do not have any categorization role. There are 5 values for the first digit:

- o 1xx: Informational - Request received, continuing process
- o 2xx: Success - The action was successfully received, understood, and accepted
- o 3xx: Redirection - Further action must be taken in order to complete the request
- o 4xx: Client Error - The request contains bad syntax or cannot be fulfilled
- o 5xx: Server Error - The server failed to fulfill an apparently valid request

The individual values of the numeric status codes defined for HTTP/1.1, and an example set of corresponding Reason-Phrase's, are presented below. The reason phrases listed here are only recommended -- they may be replaced by local equivalents without affecting the protocol.

Status-Code	=	"100"	;	Continue
		"101"	;	Switching Protocols
		"200"	;	OK
		"201"	;	Created
		"202"	;	Accepted
		"203"	;	Non-Authoritative Information
		"204"	;	No Content
		"205"	;	Reset Content
		"206"	;	Partial Content
		"300"	;	Multiple Choices
		"301"	;	Moved Permanently
		"302"	;	Moved Temporarily

"303"	: See Other
"304"	: Not Modified
"305"	: Use Proxy
"400"	: Bad Request
"401"	: Unauthorized
"402"	: Payment Required
"403"	: Forbidden
"404"	: Not Found
"405"	: Method Not Allowed
"406"	: Not Acceptable
"407"	: Proxy Authentication Required
"408"	: Request Time-out
"409"	: Conflict
"410"	: Gone
"411"	: Length Required
"412"	: Precondition Failed
"413"	: Request Entity Too Large
"414"	: Request-URI Too Large
"415"	: Unsupported Media Type
"500"	: Internal Server Error
"501"	: Not Implemented
"502"	: Bad Gateway
"503"	: Service Unavailable
"504"	: Gateway Time-out
"505"	: HTTP Version not supported

extension-code

extension-code = 3DIGIT

Reason-Phrase = *<TEXT, excluding CR, LF>

HTTP status codes are extensible. HTTP applications are not required to understand the meaning of all registered status codes, though such understanding is obviously desirable. However, applications **MUST** understand the class of any status code, as indicated by the first digit, and treat any unrecognized response as being equivalent to the x00 status code of that class, with the exception that an unrecognized response **MUST NOT** be cached. For example, if an unrecognized status code of 431 is received by the client, it can safely assume that there was something wrong with its request and treat the response as if it had received a 400 status code. In such cases, user agents **SHOULD** present to the user the entity returned with the response, since that entity is likely to include human-readable information which will explain the unusual status.

6.2 Response Header Fields

The response-header fields allow the server to pass additional information about the response which cannot be placed in the Status-Line. These header fields give information about the server and about further access to the resource identified by the Request-URI.

response-header = Age	; Section 14.6
Location	; Section 14.30
Proxy-Authenticate	; Section 14.33
Public	; Section 14.35
Retry-After	; Section 14.38
Server	; Section 14.39
Vary	; Section 14.43
Warning	; Section 14.45
WWW-Authenticate	; Section 14.46

Response-header field names can be extended reliably only in combination with a change in the protocol version. However, new or experimental header fields MAY be given the semantics of response-header fields if all parties in the communication recognize them to be response-header fields. Unrecognized header fields are treated as entity-header fields.

7 Entity

Request and Response messages MAY transfer an entity if not otherwise restricted by the request method or response status code. An entity consists of entity-header fields and an entity-body, although some responses will only include the entity-headers.

In this section, both sender and recipient refer to either the client or the server, depending on who sends and who receives the entity.

7.1 Entity Header Fields

Entity-header fields define optional metainformation about the entity-body or, if no body is present, about the resource identified by the request.

```

entity-header = Allow           ; Section 14.7
                | Content-Base  ; Section 14.11
                | Content-Encoding ; Section 14.12
                | Content-Language ; Section 14.13
                | Content-Length  ; Section 14.14
                | Content-Location ; Section 14.15
                | Content-MD5     ; Section 14.16
                | Content-Range   ; Section 14.17
                | Content-Type    ; Section 14.18
                | ETag            ; Section 14.20
                | Expires         ; Section 14.21
                | Last-Modified   ; Section 14.29
                | extension-header

```

extension-header = message-header

The extension-header mechanism allows additional entity-header fields to be defined without changing the protocol, but these fields cannot be assumed to be recognizable by the recipient. Unrecognized header fields **SHOULD** be ignored by the recipient and forwarded by proxies.

7.2 Entity Body

The entity-body (if any) sent with an HTTP request or response is in a format and encoding defined by the entity-header fields.

```
entity-body = *OCTET
```

An entity-body is only present in a message when a message-body is present, as described in section 4.3. The entity-body is obtained from the message-body by decoding any Transfer-Encoding that may have been applied to ensure safe and proper transfer of the message.

7.2.1 Type

When an entity-body is included with a message, the data type of that body is determined via the header fields Content-Type and Content-Encoding. These define a two-layer, ordered encoding model:

```
entity-body := Content-Encoding( Content-Type( data ) )
```

Content-Type specifies the media type of the underlying data. Content-Encoding may be used to indicate any additional content codings applied to the data, usually for the purpose of data compression, that are a property of the requested resource. There is no default encoding.

Any HTTP/1.1 message containing an entity-body SHOULD include a Content-Type header field defining the media type of that body. If and only if the media type is not given by a Content-Type field, the recipient MAY attempt to guess the media type via inspection of its content and/or the name extension(s) of the URL used to identify the resource. If the media type remains unknown, the recipient SHOULD treat it as type "application/octet-stream".

7.2.2 Length

The length of an entity-body is the length of the message-body after any transfer codings have been removed. Section 4.4 defines how the length of a message-body is determined.

8 Connections

8.1 Persistent Connections

8.1.1 Purpose

Prior to persistent connections, a separate TCP connection was established to fetch each URL, increasing the load on HTTP servers and causing congestion on the Internet. The use of inline images and other associated data often requires a client to make multiple requests of the same server in a short amount of time. Analyses of these performance problems are available [30][27]; analysis and results from a prototype implementation are in [26].

Persistent HTTP connections have a number of advantages:

- o By opening and closing fewer TCP connections, CPU time is saved, and memory used for TCP protocol control blocks is also saved.
- o HTTP requests and responses can be pipelined on a connection. Pipelining allows a client to make multiple requests without waiting for each response, allowing a single TCP connection to be used much more efficiently, with much lower elapsed time.
- o Network congestion is reduced by reducing the number of packets caused by TCP opens, and by allowing TCP sufficient time to determine the congestion state of the network.
- o HTTP can evolve more gracefully; since errors can be reported without the penalty of closing the TCP connection. Clients using future versions of HTTP might optimistically try a new feature, but if communicating with an older server, retry with old semantics after an error is reported.

HTTP implementations SHOULD implement persistent connections.

8.1.2 Overall Operation

A significant difference between HTTP/1.1 and earlier versions of HTTP is that persistent connections are the default behavior of any HTTP connection. That is, unless otherwise indicated, the client may assume that the server will maintain a persistent connection.

Persistent connections provide a mechanism by which a client and a server can signal the close of a TCP connection. This signaling takes place using the Connection header field. Once a close has been signaled, the client **MUST** not send any more requests on that connection.

8.1.2.1 Negotiation

An HTTP/1.1 server **MAY** assume that a HTTP/1.1 client intends to maintain a persistent connection unless a Connection header including the connection-token "close" was sent in the request. If the server chooses to close the connection immediately after sending the response, it **SHOULD** send a Connection header including the connection-token close.

An HTTP/1.1 client **MAY** expect a connection to remain open, but would decide to keep it open based on whether the response from a server contains a Connection header with the connection-token close. In case the client does not want to maintain a connection for more than that request, it **SHOULD** send a Connection header including the connection-token close.

If either the client or the server sends the close token in the Connection header, that request becomes the last one for the connection.

Clients and servers **SHOULD NOT** assume that a persistent connection is maintained for HTTP versions less than 1.1 unless it is explicitly signaled. See section 19.7.1 for more information on backwards compatibility with HTTP/1.0 clients.

In order to remain persistent, all messages on the connection must have a self-defined message length (i.e., one not defined by closure of the connection), as described in section 4.4.

8.1.2.2 Pipelining

A client that supports persistent connections **MAY** "pipeline" its requests (i.e., send multiple requests without waiting for each response). A server **MUST** send its responses to those requests in the same order that the requests were received.

Clients which assume persistent connections and pipeline immediately after connection establishment SHOULD be prepared to retry their connection if the first pipelined attempt fails. If a client does such a retry, it MUST NOT pipeline before it knows the connection is persistent. Clients MUST also be prepared to resend their requests if the server closes the connection before sending all of the corresponding responses.

8.1.3 Proxy Servers

It is especially important that proxies correctly implement the properties of the Connection header field as specified in 14.2.1.

The proxy server MUST signal persistent connections separately with its clients and the origin servers (or other proxy servers) that it connects to. Each persistent connection applies to only one transport link.

A proxy server MUST NOT establish a persistent connection with an HTTP/1.0 client.

8.1.4 Practical Considerations

Servers will usually have some time-out value beyond which they will no longer maintain an inactive connection. Proxy servers might make this a higher value since it is likely that the client will be making more connections through the same server. The use of persistent connections places no requirements on the length of this time-out for either the client or the server.

When a client or server wishes to time-out it SHOULD issue a graceful close on the transport connection. Clients and servers SHOULD both constantly watch for the other side of the transport close, and respond to it as appropriate. If a client or server does not detect the other side's close promptly it could cause unnecessary resource drain on the network.

A client, server, or proxy MAY close the transport connection at any time. For example, a client MAY have started to send a new request at the same time that the server has decided to close the "idle" connection. From the server's point of view, the connection is being closed while it was idle, but from the client's point of view, a request is in progress.

This means that clients, servers, and proxies MUST be able to recover from asynchronous close events. Client software SHOULD reopen the transport connection and retransmit the aborted request without user interaction so long as the request method is idempotent (see section

9.1.2); other methods MUST NOT be automatically retried, although user agents MAY offer a human operator the choice of retrying the request.

However, this automatic retry SHOULD NOT be repeated if the second request fails.

Servers SHOULD always respond to at least one request per connection, if at all possible. Servers SHOULD NOT close a connection in the middle of transmitting a response, unless a network or client failure is suspected.

Clients that use persistent connections SHOULD limit the number of simultaneous connections that they maintain to a given server. A single-user client SHOULD maintain AT MOST 2 connections with any server or proxy. A proxy SHOULD use up to $2*N$ connections to another server or proxy, where N is the number of simultaneously active users. These guidelines are intended to improve HTTP response times and avoid congestion of the Internet or other networks.

8.2 Message Transmission Requirements

General requirements:

- o HTTP/1.1 servers SHOULD maintain persistent connections and use TCP's flow control mechanisms to resolve temporary overloads, rather than terminating connections with the expectation that clients will retry. The latter technique can exacerbate network congestion.
- o An HTTP/1.1 (or later) client sending a message-body SHOULD monitor the network connection for an error status while it is transmitting the request. If the client sees an error status, it SHOULD immediately cease transmitting the body. If the body is being sent using a "chunked" encoding (section 3.6), a zero length chunk and empty footer MAY be used to prematurely mark the end of the message. If the body was preceded by a Content-Length header, the client MUST close the connection.
- o An HTTP/1.1 (or later) client MUST be prepared to accept a 100 (Continue) status followed by a regular response.
- o An HTTP/1.1 (or later) server that receives a request from a HTTP/1.0 (or earlier) client MUST NOT transmit the 100 (continue) response; it SHOULD either wait for the request to be completed normally (thus avoiding an interrupted request) or close the connection prematurely.

Upon receiving a method subject to these requirements from an HTTP/1.1 (or later) client, an HTTP/1.1 (or later) server MUST either respond with 100 (Continue) status and continue to read from the input stream, or respond with an error status. If it responds with an error status, it MAY close the transport (TCP) connection or it MAY continue to read and discard the rest of the request. It MUST NOT perform the requested method if it returns an error status.

Clients SHOULD remember the version number of at least the most recently used server; if an HTTP/1.1 client has seen an HTTP/1.1 or later response from the server, and it sees the connection close before receiving any status from the server, the client SHOULD retry the request without user interaction so long as the request method is idempotent (see section 9.1.2); other methods MUST NOT be automatically retried, although user agents MAY offer a human operator the choice of retrying the request.. If the client does retry the request, the client

- o MUST first send the request header fields, and then
- o MUST wait for the server to respond with either a 100 (Continue) response, in which case the client should continue, or with an error status.

If an HTTP/1.1 client has not seen an HTTP/1.1 or later response from the server, it should assume that the server implements HTTP/1.0 or older and will not use the 100 (Continue) response. If in this case the client sees the connection close before receiving any status from the server, the client SHOULD retry the request. If the client does retry the request to this HTTP/1.0 server, it should use the following "binary exponential backoff" algorithm to be assured of obtaining a reliable response:

1. Initiate a new connection to the server
2. Transmit the request-headers
3. Initialize a variable R to the estimated round-trip time to the server (e.g., based on the time it took to establish the connection), or to a constant value of 5 seconds if the round-trip time is not available.
4. Compute $T = R * (2^{**}N)$, where N is the number of previous retries of this request.
5. Wait either for an error response from the server, or for T seconds (whichever comes first)

6. If no error response is received, after T seconds transmit the body of the request.
7. If client sees that the connection is closed prematurely, repeat from step 1 until the request is accepted, an error response is received, or the user becomes impatient and terminates the retry process.

No matter what the server version, if an error status is received, the client

- o MUST NOT continue and
- o MUST close the connection if it has not completed sending the message.

An HTTP/1.1 (or later) client that sees the connection close after receiving a 100 (Continue) but before receiving any other status SHOULD retry the request, and need not wait for 100 (Continue) response (but MAY do so if this simplifies the implementation).

9 Method Definitions

The set of common methods for HTTP/1.1 is defined below. Although this set can be expanded, additional methods cannot be assumed to share the same semantics for separately extended clients and servers.

The Host request-header field (section 14.23) MUST accompany all HTTP/1.1 requests.

9.1 Safe and Idempotent Methods

9.1.1 Safe Methods

Implementers should be aware that the software represents the user in their interactions over the Internet, and should be careful to allow the user to be aware of any actions they may take which may have an unexpected significance to themselves or others.

In particular, the convention has been established that the GET and HEAD methods should never have the significance of taking an action other than retrieval. These methods should be considered "safe." This allows user agents to represent other methods, such as POST, PUT and DELETE, in a special way, so that the user is made aware of the fact that a possibly unsafe action is being requested.

Naturally, it is not possible to ensure that the server does not generate side-effects as a result of performing a GET request; in

fact, some dynamic resources consider that a feature. The important distinction here is that the user did not request the side-effects, so therefore cannot be held accountable for them.

9.1.2 Idempotent Methods

Methods may also have the property of "idempotence" in that (aside from error or expiration issues) the side-effects of $N > 0$ identical requests is the same as for a single request. The methods GET, HEAD, PUT and DELETE share this property.

9.2 OPTIONS

The OPTIONS method represents a request for information about the communication options available on the request/response chain identified by the Request-URI. This method allows the client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action or initiating a resource retrieval.

Unless the server's response is an error, the response **MUST NOT** include entity information other than what can be considered as communication options (e.g., Allow is appropriate, but Content-Type is not). Responses to this method are not cachable.

If the Request-URI is an asterisk ("*"), the OPTIONS request is intended to apply to the server as a whole. A 200 response **SHOULD** include any header fields which indicate optional features implemented by the server (e.g., Public), including any extensions not defined by this specification, in addition to any applicable general or response-header fields. As described in section 5.1.2, an "OPTIONS *" request can be applied through a proxy by specifying the destination server in the Request-URI without any path information.

If the Request-URI is not an asterisk, the OPTIONS request applies only to the options that are available when communicating with that resource. A 200 response **SHOULD** include any header fields which indicate optional features implemented by the server and applicable to that resource (e.g., Allow), including any extensions not defined by this specification, in addition to any applicable general or response-header fields. If the OPTIONS request passes through a proxy, the proxy **MUST** edit the response to exclude those options which apply to a proxy's capabilities and which are known to be unavailable through that proxy.

9.3 GET

The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

The semantics of the GET method change to a "conditional GET" if the request message includes an If-Modified-Since, If-Unmodified-Since, If-Match, If-None-Match, or If-Range header field. A conditional GET method requests that the entity be transferred only under the circumstances described by the conditional header field(s). The conditional GET method is intended to reduce unnecessary network usage by allowing cached entities to be refreshed without requiring multiple requests or transferring data already held by the client.

The semantics of the GET method change to a "partial GET" if the request message includes a Range header field. A partial GET requests that only part of the entity be transferred, as described in section 14.36. The partial GET method is intended to reduce unnecessary network usage by allowing partially-retrieved entities to be completed without transferring data already held by the client.

The response to a GET request is cachable if and only if it meets the requirements for HTTP caching described in section 13.

9.4 HEAD

The HEAD method is identical to GET except that the server MUST NOT return a message-body in the response. The metainformation contained in the HTTP headers in response to a HEAD request SHOULD be identical to the information sent in response to a GET request. This method can be used for obtaining metainformation about the entity implied by the request without transferring the entity-body itself. This method is often used for testing hypertext links for validity, accessibility, and recent modification.

The response to a HEAD request may be cachable in the sense that the information contained in the response may be used to update a previously cached entity from that resource. If the new field values indicate that the cached entity differs from the current entity (as would be indicated by a change in Content-Length, Content-MD5, ETag or Last-Modified), then the cache MUST treat the cache entry as stale.

9.5 POST

The POST method is used to request that the destination server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line. POST is designed to allow a uniform method to cover the following functions:

- o Annotation of existing resources;
- o Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles;
- o Providing a block of data, such as the result of submitting a form, to a data-handling process;
- o Extending a database through an append operation.

The actual function performed by the POST method is determined by the server and is usually dependent on the Request-URI. The posted entity is subordinate to that URI in the same way that a file is subordinate to a directory containing it, a news article is subordinate to a newsgroup to which it is posted, or a record is subordinate to a database.

The action performed by the POST method might not result in a resource that can be identified by a URI. In this case, either 200 (OK) or 204 (No Content) is the appropriate response status, depending on whether or not the response includes an entity that describes the result.

If a resource has been created on the origin server, the response SHOULD be 201 (Created) and contain an entity which describes the status of the request and refers to the new resource, and a Location header (see section 14.30).

Responses to this method are not cachable, unless the response includes appropriate Cache-Control or Expires header fields. However, the 303 (See Other) response can be used to direct the user agent to retrieve a cachable resource.

POST requests must obey the message transmission requirements set out in section 8.2.

9.6 PUT

The PUT method requests that the enclosed entity be stored under the supplied Request-URI. If the Request-URI refers to an already existing resource, the enclosed entity SHOULD be considered as a modified version of the one residing on the origin server. If the Request-URI does not point to an existing resource, and that URI is capable of being defined as a new resource by the requesting user agent, the origin server can create the resource with that URI. If a new resource is created, the origin server MUST inform the user agent via the 201 (Created) response. If an existing resource is modified, either the 200 (OK) or 204 (No Content) response codes SHOULD be sent to indicate successful completion of the request. If the resource could not be created or modified with the Request-URI, an appropriate error response SHOULD be given that reflects the nature of the problem. The recipient of the entity MUST NOT ignore any Content-* (e.g. Content-Range) headers that it does not understand or implement and MUST return a 501 (Not Implemented) response in such cases.

If the request passes through a cache and the Request-URI identifies one or more currently cached entities, those entries should be treated as stale. Responses to this method are not cachable.

The fundamental difference between the POST and PUT requests is reflected in the different meaning of the Request-URI. The URI in a POST request identifies the resource that will handle the enclosed entity. That resource may be a data-accepting process, a gateway to some other protocol, or a separate entity that accepts annotations. In contrast, the URI in a PUT request identifies the entity enclosed with the request -- the user agent knows what URI is intended and the server MUST NOT attempt to apply the request to some other resource. If the server desires that the request be applied to a different URI, it MUST send a 301 (Moved Permanently) response; the user agent MAY then make its own decision regarding whether or not to redirect the request.

A single resource MAY be identified by many different URIs. For example, an article may have a URI for identifying "the current version" which is separate from the URI identifying each particular version. In this case, a PUT request on a general URI may result in several other URIs being defined by the origin server.

HTTP/1.1 does not define how a PUT method affects the state of an origin server.

PUT requests must obey the message transmission requirements set out in section 8.2.

9.7 DELETE

The DELETE method requests that the origin server delete the resource identified by the Request-URI. This method MAY be overridden by human intervention (or other means) on the origin server. The client cannot be guaranteed that the operation has been carried out, even if the status code returned from the origin server indicates that the action has been completed successfully. However, the server SHOULD not indicate success unless, at the time the response is given, it intends to delete the resource or move it to an inaccessible location.

A successful response SHOULD be 200 (OK) if the response includes an entity describing the status, 202 (Accepted) if the action has not yet been enacted, or 204 (No Content) if the response is OK but does not include an entity.

If the request passes through a cache and the Request-URI identifies one or more currently cached entities, those entries should be treated as stale. Responses to this method are not cachable.

9.8 TRACE

The TRACE method is used to invoke a remote, application-layer loop-back of the request message. The final recipient of the request SHOULD reflect the message received back to the client as the entity-body of a 200 (OK) response. The final recipient is either the origin server or the first proxy or gateway to receive a Max-Forwards value of zero (0) in the request (see section 14.31). A TRACE request MUST NOT include an entity.

TRACE allows the client to see what is being received at the other end of the request chain and use that data for testing or diagnostic information. The value of the Via header field (section 14.44) is of particular interest, since it acts as a trace of the request chain. Use of the Max-Forwards header field allows the client to limit the length of the request chain, which is useful for testing a chain of proxies forwarding messages in an infinite loop.

If successful, the response SHOULD contain the entire request message in the entity-body, with a Content-Type of "message/http". Responses to this method MUST NOT be cached.

10 Status Code Definitions

Each Status-Code is described below, including a description of which method(s) it can follow and any metainformation required in the response.

10.1 Informational lxx

This class of status code indicates a provisional response, consisting only of the Status-Line and optional headers, and is terminated by an empty line. Since HTTP/1.0 did not define any lxx status codes, servers **MUST NOT** send a lxx response to an HTTP/1.0 client except under experimental conditions.

10.1.1 100 Continue

The client may continue with its request. This interim response is used to inform the client that the initial part of the request has been received and has not yet been rejected by the server. The client **SHOULD** continue by sending the remainder of the request or, if the request has already been completed, ignore this response. The server **MUST** send a final response after the request has been completed.

10.1.2 101 Switching Protocols

The server understands and is willing to comply with the client's request, via the Upgrade message header field (section 14.41), for a change in the application protocol being used on this connection. The server will switch protocols to those defined by the response's Upgrade header field immediately after the empty line which terminates the 101 response.

The protocol should only be switched when it is advantageous to do so. For example, switching to a newer version of HTTP is advantageous over older versions, and switching to a real-time, synchronous protocol may be advantageous when delivering resources that use such features.

10.2 Successful 2xx

This class of status code indicates that the client's request was successfully received, understood, and accepted.

10.2.1 200 OK

The request has succeeded. The information returned with the response is dependent on the method used in the request, for example:

GET an entity corresponding to the requested resource is sent in the response;

HEAD the entity-header fields corresponding to the requested resource are sent in the response without any message-body;

POST an entity describing or containing the result of the action;

TRACE an entity containing the request message as received by the end server.

10.2.2 201 Created

The request has been fulfilled and resulted in a new resource being created. The newly created resource can be referenced by the URI(s) returned in the entity of the response, with the most specific URL for the resource given by a Location header field. The origin server **MUST** create the resource before returning the 201 status code. If the action cannot be carried out immediately, the server should respond with 202 (Accepted) response instead.

10.2.3 202 Accepted

The request has been accepted for processing, but the processing has not been completed. The request **MAY** or **MAY NOT** eventually be acted upon, as it **MAY** be disallowed when processing actually takes place. There is no facility for re-sending a status code from an asynchronous operation such as this.

The 202 response is intentionally non-committal. Its purpose is to allow a server to accept a request for some other process (perhaps a batch-oriented process that is only run once per day) without requiring that the user agent's connection to the server persist until the process is completed. The entity returned with this response **SHOULD** include an indication of the request's current status and either a pointer to a status monitor or some estimate of when the user can expect the request to be fulfilled.

10.2.4 203 Non-Authoritative Information

The returned metainformation in the entity-header is not the definitive set as available from the origin server, but is gathered from a local or a third-party copy. The set presented **MAY** be a subset or superset of the original version. For example, including local annotation information about the resource **MAY** result in a superset of the metainformation known by the origin server. Use of this response code is not required and is only appropriate when the response would otherwise be 200 (OK).

10.2.5 204 No Content

The server has fulfilled the request but there is no new information to send back. If the client is a user agent, it **SHOULD NOT** change its document view from that which caused the request to be sent. This

response is primarily intended to allow input for actions to take place without causing a change to the user agent's active document view. The response MAY include new metainformation in the form of entity-headers, which SHOULD apply to the document currently in the user agent's active view.

The 204 response MUST NOT include a message-body, and thus is always terminated by the first empty line after the header fields.

10.2.6 205 Reset Content

The server has fulfilled the request and the user agent SHOULD reset the document view which caused the request to be sent. This response is primarily intended to allow input for actions to take place via user input, followed by a clearing of the form in which the input is given so that the user can easily initiate another input action. The response MUST NOT include an entity.

10.2.7 206 Partial Content

The server has fulfilled the partial GET request for the resource. The request must have included a Range header field (section 14.36) indicating the desired range. The response MUST include either a Content-Range header field (section 14.17) indicating the range included with this response, or a multipart/byteranges Content-Type including Content-Range fields for each part. If multipart/byteranges is not used, the Content-Length header field in the response MUST match the actual number of OCTETs transmitted in the message-body.

A cache that does not support the Range and Content-Range headers MUST NOT cache 206 (Partial) responses.

10.3 Redirection 3xx

This class of status code indicates that further action needs to be taken by the user agent in order to fulfill the request. The action required MAY be carried out by the user agent without interaction with the user if and only if the method used in the second request is GET or HEAD. A user agent SHOULD NOT automatically redirect a request more than 5 times, since such redirections usually indicate an infinite loop.

10.3.1 300 Multiple Choices

The requested resource corresponds to any one of a set of representations, each with its own specific location, and agent-driven negotiation information (section 12) is being provided so that the user (or user agent) can select a preferred representation and redirect its request to that location.

Unless it was a HEAD request, the response SHOULD include an entity containing a list of resource characteristics and location(s) from which the user or user agent can choose the one most appropriate. The entity format is specified by the media type given in the Content-Type header field. Depending upon the format and the capabilities of the user agent, selection of the most appropriate choice may be performed automatically. However, this specification does not define any standard for such automatic selection.

If the server has a preferred choice of representation, it SHOULD include the specific URL for that representation in the Location field; user agents MAY use the Location field value for automatic redirection. This response is cachable unless indicated otherwise.

10.3.2 301 Moved Permanently

The requested resource has been assigned a new permanent URI and any future references to this resource SHOULD be done using one of the returned URIs. Clients with link editing capabilities SHOULD automatically re-link references to the Request-URI to one or more of the new references returned by the server, where possible. This response is cachable unless indicated otherwise.

If the new URI is a location, its URL SHOULD be given by the Location field in the response. Unless the request method was HEAD, the entity of the response SHOULD contain a short hypertext note with a hyperlink to the new URI(s).

If the 301 status code is received in response to a request other than GET or HEAD, the user agent MUST NOT automatically redirect the request unless it can be confirmed by the user, since this might change the conditions under which the request was issued.

Note: When automatically redirecting a POST request after receiving a 301 status code, some existing HTTP/1.0 user agents will erroneously change it into a GET request.

10.3.3 302 Moved Temporarily

The requested resource resides temporarily under a different URI. Since the redirection may be altered on occasion, the client **SHOULD** continue to use the Request-URI for future requests. This response is only cachable if indicated by a Cache-Control or Expires header field.

If the new URI is a location, its URL **SHOULD** be given by the Location field in the response. Unless the request method was HEAD, the entity of the response **SHOULD** contain a short hypertext note with a hyperlink to the new URI(s).

If the 302 status code is received in response to a request other than GET or HEAD, the user agent **MUST NOT** automatically redirect the request unless it can be confirmed by the user, since this might change the conditions under which the request was issued.

Note: When automatically redirecting a POST request after receiving a 302 status code, some existing HTTP/1.0 user agents will erroneously change it into a GET request.

10.3.4 303 See Other

The response to the request can be found under a different URI and **SHOULD** be retrieved using a GET method on that resource. This method exists primarily to allow the output of a POST-activated script to redirect the user agent to a selected resource. The new URI is not a substitute reference for the originally requested resource. The 303 response is not cachable, but the response to the second (redirected) request **MAY** be cachable.

If the new URI is a location, its URL **SHOULD** be given by the Location field in the response. Unless the request method was HEAD, the entity of the response **SHOULD** contain a short hypertext note with a hyperlink to the new URI(s).

10.3.5 304 Not Modified

If the client has performed a conditional GET request and access is allowed, but the document has not been modified, the server **SHOULD** respond with this status code. The response **MUST NOT** contain a message-body.

The response **MUST** include the following header fields:

- o Date
- o ETag and/or Content-Location, if the header would have been sent in a 200 response to the same request
- o Expires, Cache-Control, and/or Vary, if the field-value might differ from that sent in any previous response for the same variant

If the conditional GET used a strong cache validator (see section 13.3.3), the response **SHOULD NOT** include other entity-headers. Otherwise (i.e., the conditional GET used a weak validator), the response **MUST NOT** include other entity-headers; this prevents inconsistencies between cached entity-bodies and updated headers.

If a 304 response indicates an entity not currently cached, then the cache **MUST** disregard the response and repeat the request without the conditional.

If a cache uses a received 304 response to update a cache entry, the cache **MUST** update the entry to reflect any new field values given in the response.

The 304 response **MUST NOT** include a message-body, and thus is always terminated by the first empty line after the header fields.

10.3.6 305 Use Proxy

The requested resource **MUST** be accessed through the proxy given by the Location field. The Location field gives the URL of the proxy. The recipient is expected to repeat the request via the proxy.

10.4 Client Error 4xx

The 4xx class of status code is intended for cases in which the client seems to have erred. Except when responding to a HEAD request, the server **SHOULD** include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition. These status codes are applicable to any request method. User agents **SHOULD** display any included entity to the user.

Note: If the client is sending data, a server implementation using TCP should be careful to ensure that the client acknowledges receipt of the packet(s) containing the response, before the server closes the input connection. If the client continues sending data to the server after the close, the server's TCP stack will send a reset packet to the client, which may erase the client's

unacknowledged input buffers before they can be read and interpreted by the HTTP application.

10.4.1 400 Bad Request

The request could not be understood by the server due to malformed syntax. The client SHOULD NOT repeat the request without modifications.

10.4.2 401 Unauthorized

The request requires user authentication. The response MUST include a WWW-Authenticate header field (section 14.46) containing a challenge applicable to the requested resource. The client MAY repeat the request with a suitable Authorization header field (section 14.8). If the request already included Authorization credentials, then the 401 response indicates that authorization has been refused for those credentials. If the 401 response contains the same challenge as the prior response, and the user agent has already attempted authentication at least once, then the user SHOULD be presented the entity that was given in the response, since that entity MAY include relevant diagnostic information. HTTP access authentication is explained in section 11.

10.4.3 402 Payment Required

This code is reserved for future use.

10.4.4 403 Forbidden

The server understood the request, but is refusing to fulfill it. Authorization will not help and the request SHOULD NOT be repeated. If the request method was not HEAD and the server wishes to make public why the request has not been fulfilled, it SHOULD describe the reason for the refusal in the entity. This status code is commonly used when the server does not wish to reveal exactly why the request has been refused, or when no other response is applicable.

10.4.5 404 Not Found

The server has not found anything matching the Request-URI. No indication is given of whether the condition is temporary or permanent.

If the server does not wish to make this information available to the client, the status code 403 (Forbidden) can be used instead. The 410 (Gone) status code SHOULD be used if the server knows, through some internally configurable mechanism, that an old resource is permanently unavailable and has no forwarding address.

10.4.6 405 Method Not Allowed

The method specified in the Request-Line is not allowed for the resource identified by the Request-URI. The response MUST include an Allow header containing a list of valid methods for the requested resource.

10.4.7 406 Not Acceptable

The resource identified by the request is only capable of generating response entities which have content characteristics not acceptable according to the accept headers sent in the request.

Unless it was a HEAD request, the response SHOULD include an entity containing a list of available entity characteristics and location(s) from which the user or user agent can choose the one most appropriate. The entity format is specified by the media type given in the Content-Type header field. Depending upon the format and the capabilities of the user agent, selection of the most appropriate choice may be performed automatically. However, this specification does not define any standard for such automatic selection.

Note: HTTP/1.1 servers are allowed to return responses which are not acceptable according to the accept headers sent in the request. In some cases, this may even be preferable to sending a 406 response. User agents are encouraged to inspect the headers of an incoming response to determine if it is acceptable. If the response could be unacceptable, a user agent SHOULD temporarily stop receipt of more data and query the user for a decision on further actions.

10.4.8 407 Proxy Authentication Required

This code is similar to 401 (Unauthorized), but indicates that the client MUST first authenticate itself with the proxy. The proxy MUST return a Proxy-Authenticate header field (section 14.33) containing a challenge applicable to the proxy for the requested resource. The client MAY repeat the request with a suitable Proxy-Authorization header field (section 14.34). HTTP access authentication is explained in section 11.

10.4.9 408 Request Timeout

The client did not produce a request within the time that the server was prepared to wait. The client MAY repeat the request without modifications at any later time.

10.4.10 409 Conflict

The request could not be completed due to a conflict with the current state of the resource. This code is only allowed in situations where it is expected that the user might be able to resolve the conflict and resubmit the request. The response body SHOULD include enough information for the user to recognize the source of the conflict. Ideally, the response entity would include enough information for the user or user agent to fix the problem; however, that may not be possible and is not required.

Conflicts are most likely to occur in response to a PUT request. If versioning is being used and the entity being PUT includes changes to a resource which conflict with those made by an earlier (third-party) request, the server MAY use the 409 response to indicate that it can't complete the request. In this case, the response entity SHOULD contain a list of the differences between the two versions in a format defined by the response Content-Type.

10.4.11 410 Gone

The requested resource is no longer available at the server and no forwarding address is known. This condition SHOULD be considered permanent. Clients with link editing capabilities SHOULD delete references to the Request-URI after user approval. If the server does not know, or has no facility to determine, whether or not the condition is permanent, the status code 404 (Not Found) SHOULD be used instead. This response is cachable unless indicated otherwise.

The 410 response is primarily intended to assist the task of web maintenance by notifying the recipient that the resource is intentionally unavailable and that the server owners desire that remote links to that resource be removed. Such an event is common for limited-time, promotional services and for resources belonging to individuals no longer working at the server's site. It is not necessary to mark all permanently unavailable resources as "gone" or to keep the mark for any length of time -- that is left to the discretion of the server owner.

10.4.12 411 Length Required

The server refuses to accept the request without a defined Content-Length. The client MAY repeat the request if it adds a valid Content-Length header field containing the length of the message-body in the request message.

10.4.13 412 Precondition Failed

The precondition given in one or more of the request-header fields evaluated to false when it was tested on the server. This response code allows the client to place preconditions on the current resource metainformation (header field data) and thus prevent the requested method from being applied to a resource other than the one intended.

10.4.14 413 Request Entity Too Large

The server is refusing to process a request because the request entity is larger than the server is willing or able to process. The server may close the connection to prevent the client from continuing the request.

If the condition is temporary, the server SHOULD include a Retry-After header field to indicate that it is temporary and after what time the client may try again.

10.4.15 414 Request-URI Too Long

The server is refusing to service the request because the Request-URI is longer than the server is willing to interpret. This rare condition is only likely to occur when a client has improperly converted a POST request to a GET request with long query information, when the client has descended into a URL "black hole" of redirection (e.g., a redirected URL prefix that points to a suffix of itself), or when the server is under attack by a client attempting to exploit security holes present in some servers using fixed-length buffers for reading or manipulating the Request-URI.

10.4.16 415 Unsupported Media Type

The server is refusing to service the request because the entity of the request is in a format not supported by the requested resource for the requested method.

10.5 Server Error 5xx

Response status codes beginning with the digit "5" indicate cases in which the server is aware that it has erred or is incapable of performing the request. Except when responding to a HEAD request, the server SHOULD include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition. User agents SHOULD display any included entity to the user. These response codes are applicable to any request method.

10.5.1 500 Internal Server Error

The server encountered an unexpected condition which prevented it from fulfilling the request.

10.5.2 501 Not Implemented

The server does not support the functionality required to fulfill the request. This is the appropriate response when the server does not recognize the request method and is not capable of supporting it for any resource.

10.5.3 502 Bad Gateway

The server, while acting as a gateway or proxy, received an invalid response from the upstream server it accessed in attempting to fulfill the request.

10.5.4 503 Service Unavailable

The server is currently unable to handle the request due to a temporary overloading or maintenance of the server. The implication is that this is a temporary condition which will be alleviated after some delay. If known, the length of the delay may be indicated in a Retry-After header. If no Retry-After is given, the client SHOULD handle the response as it would for a 500 response.

Note: The existence of the 503 status code does not imply that a server must use it when becoming overloaded. Some servers may wish to simply refuse the connection.

10.5.5 504 Gateway Timeout

The server, while acting as a gateway or proxy, did not receive a timely response from the upstream server it accessed in attempting to complete the request.

10.5.6 505 HTTP Version Not Supported

The server does not support, or refuses to support, the HTTP protocol version that was used in the request message. The server is indicating that it is unable or unwilling to complete the request using the same major version as the client, as described in section 3.1, other than with this error message. The response **SHOULD** contain an entity describing why that version is not supported and what other protocols are supported by that server.

11 Access Authentication

HTTP provides a simple challenge-response authentication mechanism which **MAY** be used by a server to challenge a client request and by a client to provide authentication information. It uses an extensible, case-insensitive token to identify the authentication scheme, followed by a comma-separated list of attribute-value pairs which carry the parameters necessary for achieving authentication via that scheme.

auth-scheme = token

auth-param = token "=" quoted-string

The 401 (Unauthorized) response message is used by an origin server to challenge the authorization of a user agent. This response **MUST** include a **WWW-Authenticate** header field containing at least one challenge applicable to the requested resource.

challenge = auth-scheme 1*SP realm *("," auth-param)

realm = "realm" "=" realm-value

realm-value = quoted-string

The realm attribute (case-insensitive) is required for all authentication schemes which issue a challenge. The realm value (case-sensitive), in combination with the canonical root URL (see section 5.1.2) of the server being accessed, defines the protection space. These realms allow the protected resources on a server to be partitioned into a set of protection spaces, each with its own authentication scheme and/or authorization database. The realm value is a string, generally assigned by the origin server, which may have additional semantics specific to the authentication scheme.

A user agent that wishes to authenticate itself with a server--usually, but not necessarily, after receiving a 401 or 401 response--**MAY** do so by including an **Authorization** header field with the request. The **Authorization** field value consists of credentials

containing the authentication information of the user agent for the realm of the resource being requested.

```
credentials    = basic-credentials
                  | auth-scheme #auth-param
```

The domain over which credentials can be automatically applied by a user agent is determined by the protection space. If a prior request has been authorized, the same credentials MAY be reused for all other requests within that protection space for a period of time determined by the authentication scheme, parameters, and/or user preference. Unless otherwise defined by the authentication scheme, a single protection space cannot extend outside the scope of its server.

If the server does not wish to accept the credentials sent with a request, it SHOULD return a 401 (Unauthorized) response. The response MUST include a WWW-Authenticate header field containing the (possibly new) challenge applicable to the requested resource and an entity explaining the refusal.

The HTTP protocol does not restrict applications to this simple challenge-response mechanism for access authentication. Additional mechanisms MAY be used, such as encryption at the transport level or via message encapsulation, and with additional header fields specifying authentication information. However, these additional mechanisms are not defined by this specification.

Proxies MUST be completely transparent regarding user agent authentication. That is, they MUST forward the WWW-Authenticate and Authorization headers untouched, and follow the rules found in section 14.8.

HTTP/1.1 allows a client to pass authentication information to and from a proxy via the Proxy-Authenticate and Proxy-Authorization headers.

11.1 Basic Authentication Scheme.

The "basic" authentication scheme is based on the model that the user agent must authenticate itself with a user-ID and a password for each realm. The realm value should be considered an opaque string which can only be compared for equality with other realms on that server. The server will service the request only if it can validate the user-ID and password for the protection space of the Request-URI. There are no optional authentication parameters.

Upon receipt of an unauthorized request for a URI within the protection space, the server MAY respond with a challenge like the following:

```
WWW-Authenticate: Basic realm="WallyWorld"
```

where "WallyWorld" is the string assigned by the server to identify the protection space of the Request-URI.

To receive authorization, the client sends the userid and password, separated by a single colon (":") character, within a base64 encoded string in the credentials.

```
basic-credentials = "Basic" SP basic-cookie
```

```
basic-cookie      = <base64 [7] encoding of user-pass,  
                    except not limited to 76 char/line>
```

```
user-pass        = userid ":" password
```

```
userid           = *TEXT excluding ":">
```

```
password         = *TEXT
```

Userids might be case sensitive.

If the user agent wishes to send the userid "Aladdin" and password "open sesame", it would use the following header field:

```
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

See section 15 for security considerations associated with Basic authentication.

11.2 Digest Authentication Scheme

A digest authentication for HTTP is specified in RFC 2069 [32].

12 Content Negotiation

Most HTTP responses include an entity which contains information for interpretation by a human user. Naturally, it is desirable to supply the user with the "best available" entity corresponding to the request. Unfortunately for servers and caches, not all users have the same preferences for what is "best," and not all user agents are equally capable of rendering all entity types. For that reason, HTTP has provisions for several mechanisms for "content negotiation" -- the process of selecting the best representation for a given response

when there are multiple representations available.

Note: This is not called "format negotiation" because the alternate representations may be of the same media type, but use different capabilities of that type, be in different languages, etc.

Any response containing an entity-body MAY be subject to negotiation, including error responses.

There are two kinds of content negotiation which are possible in HTTP: server-driven and agent-driven negotiation. These two kinds of negotiation are orthogonal and thus may be used separately or in combination. One method of combination, referred to as transparent negotiation, occurs when a cache uses the agent-driven negotiation information provided by the origin server in order to provide server-driven negotiation for subsequent requests.

12.1 Server-driven Negotiation

If the selection of the best representation for a response is made by an algorithm located at the server, it is called server-driven negotiation. Selection is based on the available representations of the response (the dimensions over which it can vary; e.g. language, content-coding, etc.) and the contents of particular header fields in the request message or on other information pertaining to the request (such as the network address of the client).

Server-driven negotiation is advantageous when the algorithm for selecting from among the available representations is difficult to describe to the user agent, or when the server desires to send its "best guess" to the client along with the first response (hoping to avoid the round-trip delay of a subsequent request if the "best guess" is good enough for the user). In order to improve the server's guess, the user agent MAY include request header fields (Accept, Accept-Language, Accept-Encoding, etc.) which describe its preferences for such a response.

Server-driven negotiation has disadvantages:

1. It is impossible for the server to accurately determine what might be "best" for any given user, since that would require complete knowledge of both the capabilities of the user agent and the intended use for the response (e.g., does the user want to view it on screen or print it on paper?).
2. Having the user agent describe its capabilities in every request can be both very inefficient (given that only a small percentage of responses have multiple representations) and a potential violation of

the user's privacy.

3. It complicates the implementation of an origin server and the algorithms for generating responses to a request.
4. It may limit a public cache's ability to use the same response for multiple user's requests.

HTTP/1.1 includes the following request-header fields for enabling server-driven negotiation through description of user agent capabilities and user preferences: Accept (section 14.1), Accept-Charset (section 14.2), Accept-Encoding (section 14.3), Accept-Language (section 14.4), and User-Agent (section 14.42). However, an origin server is not limited to these dimensions and MAY vary the response based on any aspect of the request, including information outside the request-header fields or within extension header fields not defined by this specification.

HTTP/1.1 origin servers MUST include an appropriate Vary header field (section 14.43) in any cachable response based on server-driven negotiation. The Vary header field describes the dimensions over which the response might vary (i.e. the dimensions over which the origin server picks its "best guess" response from multiple representations).

HTTP/1.1 public caches MUST recognize the Vary header field when it is included in a response and obey the requirements described in section 13.6 that describes the interactions between caching and content negotiation.

12.2 Agent-driven Negotiation

With agent-driven negotiation, selection of the best representation for a response is performed by the user agent after receiving an initial response from the origin server. Selection is based on a list of the available representations of the response included within the header fields (this specification reserves the field-name Alternates, as described in appendix 19.6.2.1) or entity-body of the initial response, with each representation identified by its own URI. Selection from among the representations may be performed automatically (if the user agent is capable of doing so) or manually by the user selecting from a generated (possibly hypertext) menu.

Agent-driven negotiation is advantageous when the response would vary over commonly-used dimensions (such as type, language, or encoding), when the origin server is unable to determine a user agent's capabilities from examining the request, and generally when public caches are used to distribute server load and reduce network usage.

Agent-driven negotiation suffers from the disadvantage of needing a second request to obtain the best alternate representation. This second request is only efficient when caching is used. In addition, this specification does not define any mechanism for supporting automatic selection, though it also does not prevent any such mechanism from being developed as an extension and used within HTTP/1.1.

HTTP/1.1 defines the 300 (Multiple Choices) and 406 (Not Acceptable) status codes for enabling agent-driven negotiation when the server is unwilling or unable to provide a varying response using server-driven negotiation.

12.3 Transparent Negotiation

Transparent negotiation is a combination of both server-driven and agent-driven negotiation. When a cache is supplied with a form of the list of available representations of the response (as in agent-driven negotiation) and the dimensions of variance are completely understood by the cache, then the cache becomes capable of performing server-driven negotiation on behalf of the origin server for subsequent requests on that resource.

Transparent negotiation has the advantage of distributing the negotiation work that would otherwise be required of the origin server and also removing the second request delay of agent-driven negotiation when the cache is able to correctly guess the right response.

This specification does not define any mechanism for transparent negotiation, though it also does not prevent any such mechanism from being developed as an extension and used within HTTP/1.1. An HTTP/1.1 cache performing transparent negotiation **MUST** include a Vary header field in the response (defining the dimensions of its variance) if it is cachable to ensure correct interoperation with all HTTP/1.1 clients. The agent-driven negotiation information supplied by the origin server **SHOULD** be included with the transparently negotiated response.

13 Caching in HTTP

HTTP is typically used for distributed information systems, where performance can be improved by the use of response caches. The HTTP/1.1 protocol includes a number of elements intended to make caching work as well as possible. Because these elements are inextricable from other aspects of the protocol, and because they interact with each other, it is useful to describe the basic caching design of HTTP separately from the detailed descriptions of methods,

headers, response codes, etc.

Caching would be useless if it did not significantly improve performance. The goal of caching in HTTP/1.1 is to eliminate the need to send requests in many cases, and to eliminate the need to send full responses in many other cases. The former reduces the number of network round-trips required for many operations; we use an "expiration" mechanism for this purpose (see section 13.2). The latter reduces network bandwidth requirements; we use a "validation" mechanism for this purpose (see section 13.3).

Requirements for performance, availability, and disconnected operation require us to be able to relax the goal of semantic transparency. The HTTP/1.1 protocol allows origin servers, caches, and clients to explicitly reduce transparency when necessary. However, because non-transparent operation may confuse non-expert users, and may be incompatible with certain server applications (such as those for ordering merchandise), the protocol requires that transparency be relaxed

- o only by an explicit protocol-level request when relaxed by client or origin server
- o only with an explicit warning to the end user when relaxed by cache or client

Therefore, the HTTP/1.1 protocol provides these important elements:

1. Protocol features that provide full semantic transparency when this is required by all parties.
2. Protocol features that allow an origin server or user agent to explicitly request and control non-transparent operation.
3. Protocol features that allow a cache to attach warnings to responses that do not preserve the requested approximation of semantic transparency.

A basic principle is that it must be possible for the clients to detect any potential relaxation of semantic transparency.

Note: The server, cache, or client implementer may be faced with design decisions not explicitly discussed in this specification. If a decision may affect semantic transparency, the implementer ought to err on the side of maintaining transparency unless a careful and complete analysis shows significant benefits in breaking transparency.

13.1.1 Cache Correctness

A correct cache **MUST** respond to a request with the most up-to-date response held by the cache that is appropriate to the request (see sections 13.2.5, 13.2.6, and 13.12) which meets one of the following conditions:

1. It has been checked for equivalence with what the origin server would have returned by revalidating the response with the origin server (section 13.3);
2. It is "fresh enough" (see section 13.2). In the default case, this means it meets the least restrictive freshness requirement of the client, server, and cache (see section 14.9); if the origin server so specifies, it is the freshness requirement of the origin server alone.
3. It includes a warning if the freshness demand of the client or the origin server is violated (see section 13.1.5 and 14.45).
4. It is an appropriate 304 (Not Modified), 305 (Proxy Redirect), or error (4xx or 5xx) response message.

If the cache can not communicate with the origin server, then a correct cache **SHOULD** respond as above if the response can be correctly served from the cache; if not it **MUST** return an error or

warning indicating that there was a communication failure.

If a cache receives a response (either an entire response, or a 304 (Not Modified) response) that it would normally forward to the requesting client, and the received response is no longer fresh, the cache SHOULD forward it to the requesting client without adding a new Warning (but without removing any existing Warning headers). A cache SHOULD NOT attempt to revalidate a response simply because that response became stale in transit; this might lead to an infinite loop. An user agent that receives a stale response without a Warning MAY display a warning indication to the user.

13.1.2 Warnings

Whenever a cache returns a response that is neither first-hand nor "fresh enough" (in the sense of condition 2 in section 13.1.1), it must attach a warning to that effect, using a Warning response-header. This warning allows clients to take appropriate action.

Warnings may be used for other purposes, both cache-related and otherwise. The use of a warning, rather than an error status code, distinguish these responses from true failures.

Warnings are always cachable, because they never weaken the transparency of a response. This means that warnings can be passed to HTTP/1.0 caches without danger; such caches will simply pass the warning along as an entity-header in the response.

Warnings are assigned numbers between 0 and 99. This specification defines the code numbers and meanings of each currently assigned warnings, allowing a client or cache to take automated action in some (but not all) cases.

Warnings also carry a warning text. The text may be in any appropriate natural language (perhaps based on the client's Accept headers), and include an optional indication of what character set is used.

Multiple warnings may be attached to a response (either by the origin server or by a cache), including multiple warnings with the same code number. For example, a server may provide the same warning with texts in both English and Basque.

When multiple warnings are attached to a response, it may not be practical or reasonable to display all of them to the user. This version of HTTP does not specify strict priority rules for deciding which warnings to display and in what order, but does suggest some heuristics.

The Warning header and the currently defined warnings are described in section 14.45.

13.1.3 Cache-control Mechanisms

The basic cache mechanisms in HTTP/1.1 (server-specified expiration times and validators) are implicit directives to caches. In some cases, a server or client may need to provide explicit directives to the HTTP caches. We use the Cache-Control header for this purpose.

The Cache-Control header allows a client or server to transmit a variety of directives in either requests or responses. These directives typically override the default caching algorithms. As a general rule, if there is any apparent conflict between header values, the most restrictive interpretation should be applied (that is, the one that is most likely to preserve semantic transparency). However, in some cases, Cache-Control directives are explicitly specified as weakening the approximation of semantic transparency (for example, "max-stale" or "public").

The Cache-Control directives are described in detail in section 14.9.

13.1.4 Explicit User Agent Warnings

Many user agents make it possible for users to override the basic caching mechanisms. For example, the user agent may allow the user to specify that cached entities (even explicitly stale ones) are never validated. Or the user agent might habitually add "Cache-Control: max-stale=3600" to every request. The user should have to explicitly request either non-transparent behavior, or behavior that results in abnormally ineffective caching.

If the user has overridden the basic caching mechanisms, the user agent should explicitly indicate to the user whenever this results in the display of information that might not meet the server's transparency requirements (in particular, if the displayed entity is known to be stale). Since the protocol normally allows the user agent to determine if responses are stale or not, this indication need only be displayed when this actually happens. The indication need not be a dialog box; it could be an icon (for example, a picture of a rotting fish) or some other visual indicator.

If the user has overridden the caching mechanisms in a way that would abnormally reduce the effectiveness of caches, the user agent should continually display an indication (for example, a picture of currency in flames) so that the user does not inadvertently consume excess resources or suffer from excessive latency.

13.1.5 Exceptions to the Rules and Warnings

In some cases, the operator of a cache may choose to configure it to return stale responses even when not requested by clients. This decision should not be made lightly, but may be necessary for reasons of availability or performance, especially when the cache is poorly connected to the origin server. Whenever a cache returns a stale response, it **MUST** mark it as such (using a Warning header). This allows the client software to alert the user that there may be a potential problem.

It also allows the user agent to take steps to obtain a first-hand or fresh response. For this reason, a cache **SHOULD NOT** return a stale response if the client explicitly requests a first-hand or fresh one, unless it is impossible to comply for technical or policy reasons.

13.1.6 Client-controlled Behavior

While the origin server (and to a lesser extent, intermediate caches, by their contribution to the age of a response) are the primary source of expiration information, in some cases the client may need to control a cache's decision about whether to return a cached response without validating it. Clients do this using several directives of the Cache-Control header.

A client's request may specify the maximum age it is willing to accept of an unvalidated response; specifying a value of zero forces the cache(s) to revalidate all responses. A client may also specify the minimum time remaining before a response expires. Both of these options increase constraints on the behavior of caches, and so cannot further relax the cache's approximation of semantic transparency.

A client may also specify that it will accept stale responses, up to some maximum amount of staleness. This loosens the constraints on the caches, and so may violate the origin server's specified constraints on semantic transparency, but may be necessary to support disconnected operation, or high availability in the face of poor connectivity.

13.2 Expiration Model

13.2.1 Server-Specified Expiration

HTTP caching works best when caches can entirely avoid making requests to the origin server. The primary mechanism for avoiding requests is for an origin server to provide an explicit expiration time in the future, indicating that a response may be used to satisfy subsequent requests. In other words, a cache can return a fresh

response without first contacting the server.

Our expectation is that servers will assign future explicit expiration times to responses in the belief that the entity is not likely to change, in a semantically significant way, before the expiration time is reached. This normally preserves semantic transparency, as long as the server's expiration times are carefully chosen.

The expiration mechanism applies only to responses taken from a cache and not to first-hand responses forwarded immediately to the requesting client.

If an origin server wishes to force a semantically transparent cache to validate every request, it may assign an explicit expiration time in the past. This means that the response is always stale, and so the cache **SHOULD** validate it before using it for subsequent requests. See section 14.9.4 for a more restrictive way to force revalidation.

If an origin server wishes to force any HTTP/1.1 cache, no matter how it is configured, to validate every request, it should use the "must-revalidate" Cache-Control directive (see section 14.9).

Servers specify explicit expiration times using either the Expires header, or the max-age directive of the Cache-Control header.

An expiration time cannot be used to force a user agent to refresh its display or reload a resource; its semantics apply only to caching mechanisms, and such mechanisms need only check a resource's expiration status when a new request for that resource is initiated. See section 13.13 for explanation of the difference between caches and history mechanisms.

13.2.2 Heuristic Expiration

Since origin servers do not always provide explicit expiration times, HTTP caches typically assign heuristic expiration times, employing algorithms that use other header values (such as the Last-Modified time) to estimate a plausible expiration time. The HTTP/1.1 specification does not provide specific algorithms, but does impose worst-case constraints on their results. Since heuristic expiration times may compromise semantic transparency, they should be used cautiously, and we encourage origin servers to provide explicit expiration times as much as possible.

13.2.3 Age Calculations

In order to know if a cached entry is fresh, a cache needs to know if its age exceeds its freshness lifetime. We discuss how to calculate the latter in section 13.2.4; this section describes how to calculate the age of a response or cache entry.

In this discussion, we use the term "now" to mean "the current value of the clock at the host performing the calculation." Hosts that use HTTP, but especially hosts running origin servers and caches, should use NTP [28] or some similar protocol to synchronize their clocks to a globally accurate time standard.

Also note that HTTP/1.1 requires origin servers to send a Date header with every response, giving the time at which the response was generated. We use the term "date_value" to denote the value of the Date header, in a form appropriate for arithmetic operations.

HTTP/1.1 uses the Age response-header to help convey age information between caches. The Age header value is the sender's estimate of the amount of time since the response was generated at the origin server. In the case of a cached response that has been revalidated with the origin server, the Age value is based on the time of revalidation, not of the original response.

In essence, the Age value is the sum of the time that the response has been resident in each of the caches along the path from the origin server, plus the amount of time it has been in transit along network paths.

We use the term "age_value" to denote the value of the Age header, in a form appropriate for arithmetic operations.

A response's age can be calculated in two entirely independent ways:

1. now minus date_value, if the local clock is reasonably well synchronized to the origin server's clock. If the result is negative, the result is replaced by zero.
2. age_value, if all of the caches along the response path implement HTTP/1.1.

Given that we have two independent ways to compute the age of a response when it is received, we can combine these as

$$\text{corrected_received_age} = \max(\text{now} - \text{date_value}, \text{age_value})$$

and as long as we have either nearly synchronized clocks or all-

HTTP/1.1 paths, one gets a reliable (conservative) result.

Note that this correction is applied at each HTTP/1.1 cache along the path, so that if there is an HTTP/1.0 cache in the path, the correct received age is computed as long as the receiving cache's clock is nearly in sync. We don't need end-to-end clock synchronization (although it is good to have), and there is no explicit clock synchronization step.

Because of network-imposed delays, some significant interval may pass from the time that a server generates a response and the time it is received at the next outbound cache or client. If uncorrected, this delay could result in improperly low ages.

Because the request that resulted in the returned Age value must have been initiated prior to that Age value's generation, we can correct for delays imposed by the network by recording the time at which the request was initiated. Then, when an Age value is received, it **MUST** be interpreted relative to the time the request was initiated, not the time that the response was received. This algorithm results in conservative behavior no matter how much delay is experienced. So, we compute:

$$\text{corrected_initial_age} = \text{corrected_received_age} + (\text{now} - \text{request_time})$$

where "request_time" is the time (according to the local clock) when the request that elicited this response was sent.

Summary of age calculation algorithm, when a cache receives a response:

```
/*
 * age_value
 *   is the value of Age: header received by the cache with
 *   this response.
 * date_value
 *   is the value of the origin server's Date: header
 * request_time
 *   is the (local) time when the cache made the request
 *   that resulted in this cached response
 * response_time
 *   is the (local) time when the cache received the
 *   response
 * now
 *   is the current (local) time
 */
apparent_age = max(0, response_time - date_value);
```

```
corrected_received_age = max(apparent_age, age_value);
response_delay = response_time - request_time;
corrected_initial_age = corrected_received_age + response_delay;
resident_time = now - response_time;
current_age = corrected_initial_age + resident_time;
```

When a cache sends a response, it must add to the `corrected_initial_age` the amount of time that the response was resident locally. It must then transmit this total age, using the Age header, to the next recipient cache.

Note that a client cannot reliably tell that a response is first-hand, but the presence of an Age header indicates that a response is definitely not first-hand. Also, if the Date in a response is earlier than the client's local request time, the response is probably not first-hand (in the absence of serious clock skew).

13.2.4 Expiration Calculations

In order to decide whether a response is fresh or stale, we need to compare its freshness lifetime to its age. The age is calculated as described in section 13.2.3; this section describes how to calculate the freshness lifetime, and to determine if a response has expired. In the discussion below, the values can be represented in any form appropriate for arithmetic operations.

We use the term "expires_value" to denote the value of the Expires header. We use the term "max_age_value" to denote an appropriate value of the number of seconds carried by the max-age directive of the Cache-Control header in a response (see section 14.10).

The max-age directive takes priority over Expires, so if max-age is present in a response, the calculation is simply:

$$\text{freshness_lifetime} = \text{max_age_value}$$

Otherwise, if Expires is present in the response, the calculation is:

$$\text{freshness_lifetime} = \text{expires_value} - \text{date_value}$$

Note that neither of these calculations is vulnerable to clock skew, since all of the information comes from the origin server.

If neither Expires nor Cache-Control: max-age appears in the response, and the response does not include other restrictions on caching, the cache MAY compute a freshness lifetime using a heuristic. If the value is greater than 24 hours, the cache must attach Warning 13 to any response whose age is more than 24 hours if

such warning has not already been added.

Also, if the response does have a Last-Modified time, the heuristic expiration value SHOULD be no more than some fraction of the interval since that time. A typical setting of this fraction might be 10%.

The calculation to determine if a response has expired is quite simple:

```
response_is_fresh = (freshness_lifetime > current_age)
```

13.2.5 Disambiguating Expiration Values

Because expiration values are assigned optimistically, it is possible for two caches to contain fresh values for the same resource that are different.

If a client performing a retrieval receives a non-first-hand response for a request that was already fresh in its own cache, and the Date header in its existing cache entry is newer than the Date on the new response, then the client MAY ignore the response. If so, it MAY retry the request with a "Cache-Control: max-age=0" directive (see section 14.9), to force a check with the origin server.

If a cache has two fresh responses for the same representation with different validators, it MUST use the one with the more recent Date header. This situation may arise because the cache is pooling responses from other caches, or because a client has asked for a reload or a revalidation of an apparently fresh cache entry.

13.2.6 Disambiguating Multiple Responses

Because a client may be receiving responses via multiple paths, so that some responses flow through one set of caches and other responses flow through a different set of caches, a client may receive responses in an order different from that in which the origin server sent them. We would like the client to use the most recently generated response, even if older responses are still apparently fresh.

Neither the entity tag nor the expiration value can impose an ordering on responses, since it is possible that a later response intentionally carries an earlier expiration time. However, the HTTP/1.1 specification requires the transmission of Date headers on every response, and the Date values are ordered to a granularity of one second.

When a client tries to revalidate a cache entry, and the response it receives contains a Date header that appears to be older than the one for the existing entry, then the client SHOULD repeat the request unconditionally, and include

Cache-Control: max-age=0

to force any intermediate caches to validate their copies directly with the origin server, or

Cache-Control: no-cache

to force any intermediate caches to obtain a new copy from the origin server.

If the Date values are equal, then the client may use either response (or may, if it is being extremely prudent, request a new response). Servers MUST NOT depend on clients being able to choose deterministically between responses generated during the same second, if their expiration times overlap.

13.3 Validation Model

When a cache has a stale entry that it would like to use as a response to a client's request, it first has to check with the origin server (or possibly an intermediate cache with a fresh response) to see if its cached entry is still usable. We call this "validating" the cache entry. Since we do not want to have to pay the overhead of retransmitting the full response if the cached entry is good, and we do not want to pay the overhead of an extra round trip if the cached entry is invalid, the HTTP/1.1 protocol supports the use of conditional methods.

The key protocol features for supporting conditional methods are those concerned with "cache validators." When an origin server generates a full response, it attaches some sort of validator to it, which is kept with the cache entry. When a client (user agent or proxy cache) makes a conditional request for a resource for which it has a cache entry, it includes the associated validator in the request.

The server then checks that validator against the current validator for the entity, and, if they match, it responds with a special status code (usually, 304 (Not Modified)) and no entity-body. Otherwise, it returns a full response (including entity-body). Thus, we avoid transmitting the full response if the validator matches, and we avoid an extra round trip if it does not match.

Note: the comparison functions used to decide if validators match are defined in section 13.3.3.

In HTTP/1.1, a conditional request looks exactly the same as a normal request for the same resource, except that it carries a special header (which includes the validator) that implicitly turns the method (usually, GET) into a conditional.

The protocol includes both positive and negative senses of cache-validating conditions. That is, it is possible to request either that a method be performed if and only if a validator matches or if and only if no validators match.

Note: a response that lacks a validator may still be cached, and served from cache until it expires, unless this is explicitly prohibited by a Cache-Control directive. However, a cache cannot do a conditional retrieval if it does not have a validator for the entity, which means it will not be refreshable after it expires.

13.3.1 Last-modified Dates

The Last-Modified entity-header field value is often used as a cache validator. In simple terms, a cache entry is considered to be valid if the entity has not been modified since the Last-Modified value.

13.3.2 Entity Tag Cache Validators

The ETag entity-header field value, an entity tag, provides for an "opaque" cache validator. This may allow more reliable validation in situations where it is inconvenient to store modification dates, where the one-second resolution of HTTP date values is not sufficient, or where the origin server wishes to avoid certain paradoxes that may arise from the use of modification dates.

Entity Tags are described in section 3.11. The headers used with entity tags are described in sections 14.20, 14.25, 14.26 and 14.43.

13.3.3 Weak and Strong Validators

Since both origin servers and caches will compare two validators to decide if they represent the same or different entities, one normally would expect that if the entity (the entity-body or any entity-headers) changes in any way, then the associated validator would change as well. If this is true, then we call this validator a "strong validator."

However, there may be cases when a server prefers to change the validator only on semantically significant changes, and not when

insignificant aspects of the entity change. A validator that does not always change when the resource changes is a "weak validator."

Entity tags are normally "strong validators," but the protocol provides a mechanism to tag an entity tag as "weak." One can think of a strong validator as one that changes whenever the bits of an entity changes, while a weak value changes whenever the meaning of an entity changes. Alternatively, one can think of a strong validator as part of an identifier for a specific entity, while a weak validator is part of an identifier for a set of semantically equivalent entities.

Note: One example of a strong validator is an integer that is incremented in stable storage every time an entity is changed.

An entity's modification time, if represented with one-second resolution, could be a weak validator, since it is possible that the resource may be modified twice during a single second.

Support for weak validators is optional; however, weak validators allow for more efficient caching of equivalent objects; for example, a hit counter on a site is probably good enough if it is updated every few days or weeks, and any value during that period is likely "good enough" to be equivalent.

A "use" of a validator is either when a client generates a request and includes the validator in a validating header field, or when a server compares two validators.

Strong validators are usable in any context. Weak validators are only usable in contexts that do not depend on exact equality of an entity. For example, either kind is usable for a conditional GET of a full entity. However, only a strong validator is usable for a sub-range retrieval, since otherwise the client may end up with an internally inconsistent entity.

The only function that the HTTP/1.1 protocol defines on validators is comparison. There are two validator comparison functions, depending on whether the comparison context allows the use of weak validators or not:

- o The strong comparison function: in order to be considered equal, both validators must be identical in every way, and neither may be weak.
- o The weak comparison function: in order to be considered equal, both validators must be identical in every way, but either or both of them may be tagged as "weak" without affecting the result.

The weak comparison function MAY be used for simple (non-subrange)

GET requests. The strong comparison function MUST be used in all other cases.

An entity tag is strong unless it is explicitly tagged as weak. Section 3.11 gives the syntax for entity tags.

A Last-Modified time, when used as a validator in a request, is implicitly weak unless it is possible to deduce that it is strong, using the following rules:

- o The validator is being compared by an origin server to the actual current validator for the entity and,
- o That origin server reliably knows that the associated entity did not change twice during the second covered by the presented validator.

or

- o The validator is about to be used by a client in an If-Modified-Since or If-Unmodified-Since header, because the client has a cache entry for the associated entity, and
- o That cache entry includes a Date value, which gives the time when the origin server sent the original response, and
- o The presented Last-Modified time is at least 60 seconds before the Date value.

or

- o The validator is being compared by an intermediate cache to the validator stored in its cache entry for the entity, and
- o That cache entry includes a Date value, which gives the time when the origin server sent the original response, and
- o The presented Last-Modified time is at least 60 seconds before the Date value.

This method relies on the fact that if two different responses were sent by the origin server during the same second, but both had the same Last-Modified time, then at least one of those responses would have a Date value equal to its Last-Modified time. The arbitrary 60-second limit guards against the possibility that the Date and Last-Modified values are generated from different clocks, or at somewhat different times during the preparation of the response. An implementation may use a value larger than 60 seconds, if it is believed that 60 seconds is too short.

If a client wishes to perform a sub-range retrieval on a value for which it has only a Last-Modified time and no opaque validator, it may do this only if the Last-Modified time is strong in the sense described here.

A cache or origin server receiving a cache-conditional request, other than a full-body GET request, **MUST** use the strong comparison function to evaluate the condition.

These rules allow HTTP/1.1 caches and clients to safely perform subrange retrievals on values that have been obtained from HTTP/1.0 servers.

13.3.4 Rules for When to Use Entity Tags and Last-modified Dates

We adopt a set of rules and recommendations for origin servers, clients, and caches regarding when various validator types should be used, and for what purposes.

HTTP/1.1 origin servers:

- o **SHOULD** send an entity tag validator unless it is not feasible to generate one.
- o **MAY** send a weak entity tag instead of a strong entity tag, if performance considerations support the use of weak entity tags, or if it is unfeasible to send a strong entity tag.
- o **SHOULD** send a Last-Modified value if it is feasible to send one, unless the risk of a breakdown in semantic transparency that could result from using this date in an If-Modified-Since header would lead to serious problems.

In other words, the preferred behavior for an HTTP/1.1 origin server is to send both a strong entity tag and a Last-Modified value.

In order to be legal, a strong entity tag **MUST** change whenever the associated entity value changes in any way. A weak entity tag **SHOULD** change whenever the associated entity changes in a semantically significant way.

Note: in order to provide semantically transparent caching, an origin server must avoid reusing a specific strong entity tag value for two different entities, or reusing a specific weak entity tag value for two semantically different entities. Cache entries may persist for arbitrarily long periods, regardless of expiration times, so it may be inappropriate to expect that a cache will never again attempt to validate an entry using a validator that it obtained at some point in the past.

HTTP/1.1 clients:

- o If an entity tag has been provided by the origin server, **MUST** use that entity tag in any cache-conditional request (using If-Match or If-None-Match).

- o If only a Last-Modified value has been provided by the origin server, SHOULD use that value in non-subrange cache-conditional requests (using If-Modified-Since).
- o If only a Last-Modified value has been provided by an HTTP/1.0 origin server, MAY use that value in subrange cache-conditional requests (using If-Unmodified-Since:). The user agent should provide a way to disable this, in case of difficulty.
- o If both an entity tag and a Last-Modified value have been provided by the origin server, SHOULD use both validators in cache-conditional requests. This allows both HTTP/1.0 and HTTP/1.1 caches to respond appropriately.

An HTTP/1.1 cache, upon receiving a request, MUST use the most restrictive validator when deciding whether the client's cache entry matches the cache's own cache entry. This is only an issue when the request contains both an entity tag and a last-modified-date validator (If-Modified-Since or If-Unmodified-Since).

A note on rationale: The general principle behind these rules is that HTTP/1.1 servers and clients should transmit as much non-redundant information as is available in their responses and requests. HTTP/1.1 systems receiving this information will make the most conservative assumptions about the validators they receive.

HTTP/1.0 clients and caches will ignore entity tags. Generally, last-modified values received or used by these systems will support transparent and efficient caching, and so HTTP/1.1 origin servers should provide Last-Modified values. In those rare cases where the use of a Last-Modified value as a validator by an HTTP/1.0 system could result in a serious problem, then HTTP/1.1 origin servers should not provide one.

13.3.5 Non-validating Conditionals

The principle behind entity tags is that only the service author knows the semantics of a resource well enough to select an appropriate cache validation mechanism, and the specification of any validator comparison function more complex than byte-equality would open up a can of worms. Thus, comparisons of any other headers (except Last-Modified, for compatibility with HTTP/1.0) are never used for purposes of validating a cache entry.

13.4 Response Cachability

Unless specifically constrained by a Cache-Control (section 14.9) directive, a caching system may always store a successful response (see section 13.8) as a cache entry, may return it without validation if it is fresh, and may return it after successful validation. If

there is neither a cache validator nor an explicit expiration time associated with a response, we do not expect it to be cached, but certain caches may violate this expectation (for example, when little or no network connectivity is available). A client can usually detect that such a response was taken from a cache by comparing the Date header to the current time.

Note that some HTTP/1.0 caches are known to violate this expectation without providing any Warning.

However, in some cases it may be inappropriate for a cache to retain an entity, or to return it in response to a subsequent request. This may be because absolute semantic transparency is deemed necessary by the service author, or because of security or privacy considerations. Certain Cache-Control directives are therefore provided so that the server can indicate that certain resource entities, or portions thereof, may not be cached regardless of other considerations.

Note that section 14.8 normally prevents a shared cache from saving and returning a response to a previous request if that request included an Authorization header.

A response received with a status code of 200, 203, 206, 300, 301 or 410 may be stored by a cache and used in reply to a subsequent request, subject to the expiration mechanism, unless a Cache-Control directive prohibits caching. However, a cache that does not support the Range and Content-Range headers MUST NOT cache 206 (Partial Content) responses.

A response received with any other status code MUST NOT be returned in a reply to a subsequent request unless there are Cache-Control directives or another header(s) that explicitly allow it. For example, these include the following: an Expires header (section 14.21); a "max-age", "must-revalidate", "proxy-revalidate", "public" or "private" Cache-Control directive (section 14.9).

13.5 Constructing Responses From Caches

The purpose of an HTTP cache is to store information received in response to requests, for use in responding to future requests. In many cases, a cache simply returns the appropriate parts of a response to the requester. However, if the cache holds a cache entry based on a previous response, it may have to combine parts of a new response with what is held in the cache entry.

13.5.1 End-to-end and Hop-by-hop Headers

For the purpose of defining the behavior of caches and non-caching proxies, we divide HTTP headers into two categories:

- o End-to-end headers, which must be transmitted to the ultimate recipient of a request or response. End-to-end headers in responses must be stored as part of a cache entry and transmitted in any response formed from a cache entry.
- o Hop-by-hop headers, which are meaningful only for a single transport-level connection, and are not stored by caches or forwarded by proxies.

The following HTTP/1.1 headers are hop-by-hop headers:

- o Connection
- o Keep-Alive
- o Public
- o Proxy-Authenticate
- o Transfer-Encoding
- o Upgrade

All other headers defined by HTTP/1.1 are end-to-end headers.

Hop-by-hop headers introduced in future versions of HTTP **MUST** be listed in a Connection header, as described in section 14.10.

13.5.2 Non-modifiable Headers

Some features of the HTTP/1.1 protocol, such as Digest Authentication, depend on the value of certain end-to-end headers. A cache or non-caching proxy **SHOULD NOT** modify an end-to-end header unless the definition of that header requires or specifically allows that.

A cache or non-caching proxy **MUST NOT** modify any of the following fields in a request or response, nor may it add any of these fields if not already present:

- o Content-Location
- o ETag
- o Expires
- o Last-Modified

A cache or non-caching proxy **MUST NOT** modify or add any of the following fields in a response that contains the no-transform Cache-Control directive, or in any request:

- o Content-Encoding
- o Content-Length
- o Content-Range
- o Content-Type

A cache or non-caching proxy **MAY** modify or add these fields in a response that does not include no-transform, but if it does so, it **MUST** add a Warning 14 (Transformation applied) if one does not already appear in the response.

Warning: unnecessary modification of end-to-end headers may cause authentication failures if stronger authentication mechanisms are introduced in later versions of HTTP. Such authentication mechanisms may rely on the values of header fields not listed here.

13.5.3 Combining Headers

When a cache makes a validating request to a server, and the server provides a 304 (Not Modified) response, the cache must construct a response to send to the requesting client. The cache uses the entity-body stored in the cache entry as the entity-body of this outgoing response. The end-to-end headers stored in the cache entry are used for the constructed response, except that any end-to-end headers provided in the 304 response **MUST** replace the corresponding headers from the cache entry. Unless the cache decides to remove the cache entry, it **MUST** also replace the end-to-end headers stored with the cache entry with corresponding headers received in the incoming response.

In other words, the set of end-to-end headers received in the incoming response overrides all corresponding end-to-end headers stored with the cache entry. The cache may add Warning headers (see section 14.45) to this set.

If a header field-name in the incoming response matches more than one header in the cache entry, all such old headers are replaced.

Note: this rule allows an origin server to use a 304 (Not Modified) response to update any header associated with a previous response for the same entity, although it might not always be meaningful or correct to do so. This rule does not allow an origin server to use a 304 (not Modified) response to entirely delete a header that it had provided with a previous response.

13.5.4 Combining Byte Ranges

A response may transfer only a subrange of the bytes of an entity-body, either because the request included one or more Range specifications, or because a connection was broken prematurely. After several such transfers, a cache may have received several ranges of the same entity-body.

If a cache has a stored non-empty set of subranges for an entity, and an incoming response transfers another subrange, the cache MAY combine the new subrange with the existing set if both the following conditions are met:

- o Both the incoming response and the cache entry must have a cache validator.
- o The two cache validators must match using the strong comparison function (see section 13.3.3).

If either requirement is not meant, the cache must use only the most recent partial response (based on the Date values transmitted with every response, and using the incoming response if these values are equal or missing), and must discard the other partial information.

13.6 Caching Negotiated Responses

Use of server-driven content negotiation (section 12), as indicated by the presence of a Vary header field in a response, alters the conditions and procedure by which a cache can use the response for subsequent requests.

A server MUST use the Vary header field (section 14.43) to inform a cache of what header field dimensions are used to select among multiple representations of a cachable response. A cache may use the selected representation (the entity included with that particular response) for replying to subsequent requests on that resource only when the subsequent requests have the same or equivalent values for all header fields specified in the Vary response-header. Requests with a different value for one or more of those header fields would be forwarded toward the origin server.

If an entity tag was assigned to the representation, the forwarded request SHOULD be conditional and include the entity tags in an If-None-Match header field from all its cache entries for the Request-URI. This conveys to the server the set of entities currently held by the cache, so that if any one of these entities matches the requested entity, the server can use the ETag header in its 304 (Not Modified) response to tell the cache which entry is appropriate. If the entity-tag of the new response matches that of an existing entry, the

new response SHOULD be used to update the header fields of the existing entry, and the result MUST be returned to the client.

The Vary header field may also inform the cache that the representation was selected using criteria not limited to the request-headers; in this case, a cache MUST NOT use the response in a reply to a subsequent request unless the cache relays the new request to the origin server in a conditional request and the server responds with 304 (Not Modified), including an entity tag or Content-Location that indicates which entity should be used.

If any of the existing cache entries contains only partial content for the associated entity, its entity-tag SHOULD NOT be included in the If-None-Match header unless the request is for a range that would be fully satisfied by that entry.

If a cache receives a successful response whose Content-Location field matches that of an existing cache entry for the same Request-URI, whose entity-tag differs from that of the existing entry, and whose Date is more recent than that of the existing entry, the existing entry SHOULD NOT be returned in response to future requests, and should be deleted from the cache.

13.7 Shared and Non-Shared Caches

For reasons of security and privacy, it is necessary to make a distinction between "shared" and "non-shared" caches. A non-shared cache is one that is accessible only to a single user. Accessibility in this case SHOULD be enforced by appropriate security mechanisms. All other caches are considered to be "shared." Other sections of this specification place certain constraints on the operation of shared caches in order to prevent loss of privacy or failure of access controls.

13.8 Errors or Incomplete Response Cache Behavior

A cache that receives an incomplete response (for example, with fewer bytes of data than specified in a Content-Length header) may store the response. However, the cache MUST treat this as a partial response. Partial responses may be combined as described in section 13.5.4; the result might be a full response or might still be partial. A cache MUST NOT return a partial response to a client without explicitly marking it as such, using the 206 (Partial Content) status code. A cache MUST NOT return a partial response using a status code of 200 (OK).

If a cache receives a 5xx response while attempting to revalidate an entry, it may either forward this response to the requesting client,

or act as if the server failed to respond. In the latter case, it MAY return a previously received response unless the cached entry includes the "must-revalidate" Cache-Control directive (see section 14.9).

13.9 Side Effects of GET and HEAD

Unless the origin server explicitly prohibits the caching of their responses, the application of GET and HEAD methods to any resources SHOULD NOT have side effects that would lead to erroneous behavior if these responses are taken from a cache. They may still have side effects, but a cache is not required to consider such side effects in its caching decisions. Caches are always expected to observe an origin server's explicit restrictions on caching.

We note one exception to this rule: since some applications have traditionally used GETs and HEADs with query URLs (those containing a "?" in the rel_path part) to perform operations with significant side effects, caches MUST NOT treat responses to such URLs as fresh unless the server provides an explicit expiration time. This specifically means that responses from HTTP/1.0 servers for such URIs should not be taken from a cache. See section 9.1.1 for related information.

13.10 Invalidation After Updates or Deletions

The effect of certain methods at the origin server may cause one or more existing cache entries to become non-transparently invalid. That is, although they may continue to be "fresh," they do not accurately reflect what the origin server would return for a new request.

There is no way for the HTTP protocol to guarantee that all such cache entries are marked invalid. For example, the request that caused the change at the origin server may not have gone through the proxy where a cache entry is stored. However, several rules help reduce the likelihood of erroneous behavior.

In this section, the phrase "invalidate an entity" means that the cache should either remove all instances of that entity from its storage, or should mark these as "invalid" and in need of a mandatory revalidation before they can be returned in response to a subsequent request.

Some HTTP methods may invalidate an entity. This is either the entity referred to by the Request-URI, or by the Location or Content-Location response-headers (if present). These methods are:

- o PUT
- o DELETE
- o POST

In order to prevent denial of service attacks, an invalidation based on the URI in a Location or Content-Location header **MUST** only be performed if the host part is the same as in the Request-URI.

13.11 Write-Through Mandatory

All methods that may be expected to cause modifications to the origin server's resources **MUST** be written through to the origin server. This currently includes all methods except for GET and HEAD. A cache **MUST NOT** reply to such a request from a client before having transmitted the request to the inbound server, and having received a corresponding response from the inbound server. This does not prevent a cache from sending a 100 (Continue) response before the inbound server has replied.

The alternative (known as "write-back" or "copy-back" caching) is not allowed in HTTP/1.1, due to the difficulty of providing consistent updates and the problems arising from server, cache, or network failure prior to write-back.

13.12 Cache Replacement

If a new cachable (see sections 14.9.2, 13.2.5, 13.2.6 and 13.8) response is received from a resource while any existing responses for the same resource are cached, the cache **SHOULD** use the new response to reply to the current request. It may insert it into cache storage and may, if it meets all other requirements, use it to respond to any future requests that would previously have caused the old response to be returned. If it inserts the new response into cache storage it should follow the rules in section 13.5.3.

Note: a new response that has an older Date header value than existing cached responses is not cachable.

13.13 History Lists

User agents often have history mechanisms, such as "Back" buttons and history lists, which can be used to redisplay an entity retrieved earlier in a session.

History mechanisms and caches are different. In particular history mechanisms SHOULD NOT try to show a semantically transparent view of the current state of a resource. Rather, a history mechanism is meant to show exactly what the user saw at the time when the resource was retrieved.

By default, an expiration time does not apply to history mechanisms. If the entity is still in storage, a history mechanism should display it even if the entity has expired, unless the user has specifically configured the agent to refresh expired history documents.

This should not be construed to prohibit the history mechanism from telling the user that a view may be stale.

Note: if history list mechanisms unnecessarily prevent users from viewing stale resources, this will tend to force service authors to avoid using HTTP expiration controls and cache controls when they would otherwise like to. Service authors may consider it important that users not be presented with error messages or warning messages when they use navigation controls (such as BACK) to view previously fetched resources. Even though sometimes such resources ought not to be cached, or ought to expire quickly, user interface considerations may force service authors to resort to other means of preventing caching (e.g. "once-only" URLs) in order not to suffer the effects of improperly functioning history mechanisms.

14 Header Field Definitions

This section defines the syntax and semantics of all standard HTTP/1.1 header fields. For entity-header fields, both sender and recipient refer to either the client or the server, depending on who sends and who receives the entity.

14.1 Accept

The Accept request-header field can be used to specify certain media types which are acceptable for the response. Accept headers can be used to indicate that the request is specifically limited to a small set of desired types, as in the case of a request for an in-line image.

```
Accept          = "Accept" ":"
                  #( media-range [ accept-params ] )

media-range     = ( "*"/*
                  | ( type "/" "*" )
                  | ( type "/" subtype )
                  ) *( ";" parameter )

accept-params   = ";" "q" "=" qvalue *( accept-extension )

accept-extension = ";" token [ "=" ( token | quoted-string ) ]
```

The asterisk "*" character is used to group media types into ranges, with "*"/* indicating all media types and "type/*" indicating all subtypes of that type. The media-range MAY include media type parameters that are applicable to that range.

Each media-range MAY be followed by one or more accept-params, beginning with the "q" parameter for indicating a relative quality factor. The first "q" parameter (if any) separates the media-range parameter(s) from the accept-params. Quality factors allow the user or user agent to indicate the relative degree of preference for that media-range, using the qvalue scale from 0 to 1 (section 3.9). The default value is q=1.

Note: Use of the "q" parameter name to separate media type parameters from Accept extension parameters is due to historical practice. Although this prevents any media type parameter named "q" from being used with a media range, such an event is believed to be unlikely given the lack of any "q" parameters in the IANA media type registry and the rare usage of any media type parameters in Accept. Future media types should be discouraged from registering any parameter named "q".

The example

```
Accept: audio/*; q=0.2, audio/basic
```

SHOULD be interpreted as "I prefer audio/basic, but send me any audio type if it is the best available after an 80% mark-down in quality."

If no Accept header field is present, then it is assumed that the client accepts all media types. If an Accept header field is present, and if the server cannot send a response which is acceptable according to the combined Accept field value, then the server SHOULD send a 406 (not acceptable) response.

A more elaborate example is

```
Accept: text/plain; q=0.5, text/html,  
       text/x-dvi; q=0.8, text/x-c
```

Verbally, this would be interpreted as "text/html and text/x-c are the preferred media types, but if they do not exist, then send the text/x-dvi entity, and if that does not exist, send the text/plain entity."

Media ranges can be overridden by more specific media ranges or specific media types. If more than one media range applies to a given type, the most specific reference has precedence. For example,

```
Accept: text/*; text/html; text/html;level=1, */*
```

have the following precedence:

- 1) text/html;level=1
- 2) text/html
- 3) text/*
- 4) */*

The media type quality factor associated with a given type is determined by finding the media range with the highest precedence which matches that type. For example,

```
Accept: text/*;q=0.3, text/html;q=0.7, text/html;level=1,  
       text/html;level=2;q=0.4, */*;q=0.5
```

would cause the following values to be associated:

text/html;level=1	= 1
text/html	= 0.7
text/plain	= 0.3
image/jpeg	= 0.5
text/html;level=2	= 0.4
text/html;level=3	= 0.7

Note: A user agent may be provided with a default set of quality values for certain media ranges. However, unless the user agent is a closed system which cannot interact with other rendering agents,

this default set should be configurable by the user.

14.2 Accept-Charset

The Accept-Charset request-header field can be used to indicate what character sets are acceptable for the response. This field allows clients capable of understanding more comprehensive or special-purpose character sets to signal that capability to a server which is capable of representing documents in those character sets. The ISO-8859-1 character set can be assumed to be acceptable to all user agents.

```
Accept-Charset = "Accept-Charset" ":"  
                1#( charset [ ";" "q" "=" qvalue. ] )
```

Character set values are described in section 3.4. Each charset may be given an associated quality value which represents the user's preference for that charset. The default value is q=1. An example is

```
Accept-Charset: iso-8859-5, unicode-1-1;q=0.8
```

If no Accept-Charset header is present, the default is that any character set is acceptable. If an Accept-Charset header is present, and if the server cannot send a response which is acceptable according to the Accept-Charset header, then the server **SHOULD** send an error response with the 406 (not acceptable) status code, though the sending of an unacceptable response is also allowed.

14.3 Accept-Encoding

The Accept-Encoding request-header field is similar to Accept, but restricts the content-coding values (section 14.12) which are acceptable in the response.

```
Accept-Encoding = "Accept-Encoding" ":"  
                 #( content-coding )
```

An example of its use is

```
Accept-Encoding: compress, gzip
```

If no Accept-Encoding header is present in a request, the server **MAY** assume that the client will accept any content coding. If an Accept-Encoding header is present, and if the server cannot send a response which is acceptable according to the Accept-Encoding header, then the server **SHOULD** send an error response with the 406 (Not Acceptable) status code.

An empty Accept-Encoding value indicates none are acceptable.

14.4 Accept-Language

The Accept-Language request-header field is similar to Accept, but restricts the set of natural languages that are preferred as a response to the request.

```
Accept-Language = "Accept-Language" ":"
                  1#( language-range [ ";" "q" "=" qvalue ] )

language-range = ( ( 1*8ALPHA *( "-" 1*8ALPHA ) ) | "*" )
```

Each language-range MAY be given an associated quality value which represents an estimate of the user's preference for the languages specified by that range. The quality value defaults to "q=1". For example,

```
Accept-Language: da, en-gb;q=0.8, en;q=0.7
```

would mean: "I prefer Danish, but will accept British English and other types of English." A language-range matches a language-tag if it exactly equals the tag, or if it exactly equals a prefix of the tag such that the first tag character following the prefix is "-". The special range "*", if present in the Accept-Language field, matches every tag not matched by any other range present in the Accept-Language field.

Note: This use of a prefix matching rule does not imply that language tags are assigned to languages in such a way that it is always true that if a user understands a language with a certain tag, then this user will also understand all languages with tags for which this tag is a prefix. The prefix rule simply allows the use of prefix tags if this is the case.

The language quality factor assigned to a language-tag by the Accept-Language field is the quality value of the longest language-range in the field that matches the language-tag. If no language-range in the field matches the tag, the language quality factor assigned is 0. If no Accept-Language header is present in the request, the server SHOULD assume that all languages are equally acceptable. If an Accept-Language header is present, then all languages which are assigned a quality factor greater than 0 are acceptable.

It may be contrary to the privacy expectations of the user to send an Accept-Language header with the complete linguistic preferences of the user in every request. For a discussion of this issue, see

section 15.7.

Note: As intelligibility is highly dependent on the individual user, it is recommended that client applications make the choice of linguistic preference available to the user. If the choice is not made available, then the Accept-Language header field must not be given in the request.

14.5 Accept-Ranges

The Accept-Ranges response-header field allows the server to indicate its acceptance of range requests for a resource:

Accept-Ranges = "Accept-Ranges" ":" acceptable-ranges

acceptable-ranges = 1#range-unit | "none"

Origin servers that accept byte-range requests MAY send

Accept-Ranges: bytes

but are not required to do so. Clients MAY generate byte-range requests without having received this header for the resource involved.

Servers that do not accept any kind of range request for a resource MAY send

Accept-Ranges: none

to advise the client not to attempt a range request.

14.6 Age

The Age response-header field conveys the sender's estimate of the amount of time since the response (or its revalidation) was generated at the origin server. A cached response is "fresh" if its age does not exceed its freshness lifetime. Age values are calculated as specified in section 13.2.3.

Age = "Age" ":" age-value

age-value = delta-seconds

Age values are non-negative decimal integers, representing time in seconds.

If a cache receives a value larger than the largest positive integer it can represent, or if any of its age calculations overflows, it MUST transmit an Age header with a value of 2147483648 (2^{31}). HTTP/1.1 caches MUST send an Age header in every response. Caches SHOULD use an arithmetic type of at least 31 bits of range.

14.7 Allow

The Allow entity-header field lists the set of methods supported by the resource identified by the Request-URI. The purpose of this field is strictly to inform the recipient of valid methods associated with the resource. An Allow header field MUST be present in a 405 (Method Not Allowed) response.

Allow = "Allow" ":" 1#method

Example of use:

Allow: GET, HEAD, PUT

This field cannot prevent a client from trying other methods. However, the indications given by the Allow header field value SHOULD be followed. The actual set of allowed methods is defined by the origin server at the time of each request.

The Allow header field MAY be provided with a PUT request to recommend the methods to be supported by the new or modified resource. The server is not required to support these methods and SHOULD include an Allow header in the response giving the actual supported methods.

A proxy MUST NOT modify the Allow header field even if it does not understand all the methods specified, since the user agent MAY have other means of communicating with the origin server.

The Allow header field does not indicate what methods are implemented at the server level. Servers MAY use the Public response-header field (section 14.35) to describe what methods are implemented on the server as a whole.

14.8 Authorization

A user agent that wishes to authenticate itself with a server--usually, but not necessarily, after receiving a 401 response--MAY do so by including an Authorization request-header field with the request. The Authorization field value consists of credentials containing the authentication information of the user agent for the realm of the resource being requested.

Authorization = "Authorization" ":" credentials

HTTP access authentication is described in section 11. If a request is authenticated and a realm specified, the same credentials **SHOULD** be valid for all other requests within this realm.

When a shared cache (see section 13.7) receives a request containing an Authorization field, it **MUST NOT** return the corresponding response as a reply to any other request, unless one of the following specific exceptions holds:

1. If the response includes the "proxy-revalidate" Cache-Control directive, the cache **MAY** use that response in replying to a subsequent request, but a proxy cache **MUST** first revalidate it with the origin server, using the request-headers from the new request to allow the origin server to authenticate the new request.
2. If the response includes the "must-revalidate" Cache-Control directive, the cache **MAY** use that response in replying to a subsequent request, but all caches **MUST** first revalidate it with the origin server, using the request-headers from the new request to allow the origin server to authenticate the new request.
3. If the response includes the "public" Cache-Control directive, it may be returned in reply to any subsequent request.

14.9 Cache-Control

The Cache-Control general-header field is used to specify directives that **MUST** be obeyed by all caching mechanisms along the request/response chain. The directives specify behavior intended to prevent caches from adversely interfering with the request or response. These directives typically override the default caching algorithms. Cache directives are unidirectional in that the presence of a directive in a request does not imply that the same directive should be given in the response.

Note that HTTP/1.0 caches may not implement Cache-Control and may only implement Pragma: no-cache (see section 14.32).

Cache directives must be passed through by a proxy or gateway application, regardless of their significance to that application, since the directives may be applicable to all recipients along the request/response chain. It is not possible to specify a cache-directive for a specific cache.

Cache-Control = "Cache-Control" ":" 1#cache-directive

cache-directive = cache-request-directive
| cache-response-directive

```

cache-request-directive =
    "no-cache" [ "=" <"> 1#field-name <"> ]
    "no-store"
    "max-age" "=" delta-seconds
    "max-stale" [ "=" delta-seconds ]
    "min-fresh" "=" delta-seconds
    "only-if-cached"
    cache-extension

cache-response-directive =
    "public"
    "private" [ "=" <"> 1#field-name <"> ]
    "no-cache" [ "=" <"> 1#field-name <"> ]
    "no-store"
    "no-transform"
    "must-revalidate"
    "proxy-revalidate"
    "max-age" "=" delta-seconds
    cache-extension

cache-extension = token [ "=" ( token | quoted-string ) ]

```

When a directive appears without any `1#field-name` parameter, the directive applies to the entire request or response. When such a directive appears with a `1#field-name` parameter, it applies only to the named field or fields, and not to the rest of the request or response. This mechanism supports extensibility; implementations of future versions of the HTTP protocol may apply these directives to header fields not defined in HTTP/1.1.

The cache-control directives can be broken down into these general categories:

- o Restrictions on what is cachable; these may only be imposed by the origin server.
- o Restrictions on what may be stored by a cache; these may be imposed by either the origin server or the user agent.
- o Modifications of the basic expiration mechanism; these may be imposed by either the origin server or the user agent.
- o Controls over cache revalidation and reload; these may only be imposed by a user agent.
- o Control over transformation of entities.
- o Extensions to the caching system.

14.9.1 What is Cachable

By default, a response is cachable if the requirements of the request method, request header fields, and the response status indicate that it is cachable. Section 13.4 summarizes these defaults for cachability. The following Cache-Control response directives allow an origin server to override the default cachability of a response:

public

Indicates that the response is cachable by any cache, even if it would normally be non-cachable or cachable only within a non-shared cache. (See also Authorization, section 14.8, for additional details.)

private

Indicates that all or part of the response message is intended for a single user and MUST NOT be cached by a shared cache. This allows an origin server to state that the specified parts of the response are intended for only one user and are not a valid response for requests by other users. A private (non-shared) cache may cache the response.

Note: This usage of the word private only controls where the response may be cached, and cannot ensure the privacy of the message content.

no-cache

Indicates that all or part of the response message MUST NOT be cached anywhere. This allows an origin server to prevent caching even by caches that have been configured to return stale responses to client requests.

Note: Most HTTP/1.0 caches will not recognize or obey this directive.

14.9.2 What May be Stored by Caches

The purpose of the no-store directive is to prevent the inadvertent release or retention of sensitive information (for example, on backup tapes). The no-store directive applies to the entire message, and may be sent either in a response or in a request. If sent in a request, a cache MUST NOT store any part of either this request or any response to it. If sent in a response, a cache MUST NOT store any part of either this response or the request that elicited it. This directive applies to both non-shared and shared caches. "MUST NOT store" in this context means that the cache MUST NOT intentionally store the information in non-volatile storage, and MUST make a best-effort attempt to remove the information from volatile storage as promptly as possible after forwarding it.

Even when this directive is associated with a response, users may explicitly store such a response outside of the caching system (e.g., with a "Save As" dialog). History buffers may store such responses as part of their normal operation.

The purpose of this directive is to meet the stated requirements of certain users and service authors who are concerned about accidental releases of information via unanticipated accesses to cache data structures. While the use of this directive may improve privacy in some cases, we caution that it is NOT in any way a reliable or sufficient mechanism for ensuring privacy. In particular, malicious or compromised caches may not recognize or obey this directive; and communications networks may be vulnerable to eavesdropping.

14.9.3 Modifications of the Basic Expiration Mechanism

The expiration time of an entity may be specified by the origin server using the Expires header (see section 14.21). Alternatively, it may be specified using the max-age directive in a response.

If a response includes both an Expires header and a max-age directive, the max-age directive overrides the Expires header, even if the Expires header is more restrictive. This rule allows an origin server to provide, for a given response, a longer expiration time to an HTTP/1.1 (or later) cache than to an HTTP/1.0 cache. This may be useful if certain HTTP/1.0 caches improperly calculate ages or expiration times, perhaps due to desynchronized clocks.

Note: most older caches, not compliant with this specification, do not implement any Cache-Control directives. An origin server wishing to use a Cache-Control directive that restricts, but does not prevent, caching by an HTTP/1.1-compliant cache may exploit the requirement that the max-age directive overrides the Expires header, and the fact that non-HTTP/1.1-compliant caches do not observe the max-age directive.

Other directives allow an user agent to modify the basic expiration mechanism. These directives may be specified on a request:

max-age

Indicates that the client is willing to accept a response whose age is no greater than the specified time in seconds. Unless max-stale directive is also included, the client is not willing to accept a stale response.

min-fresh

Indicates that the client is willing to accept a response whose freshness lifetime is no less than its current age plus the

specified time in seconds. That is, the client wants a response that will still be fresh for at least the specified number of seconds.

max-stale

Indicates that the client is willing to accept a response that has exceeded its expiration time. If max-stale is assigned a value, then the client is willing to accept a response that has exceeded its expiration time by no more than the specified number of seconds. If no value is assigned to max-stale, then the client is willing to accept a stale response of any age.

If a cache returns a stale response, either because of a max-stale directive on a request, or because the cache is configured to override the expiration time of a response, the cache **MUST** attach a Warning header to the stale response, using Warning 10 (Response is stale).

14.9.4 Cache Revalidation and Reload Controls

Sometimes an user agent may want or need to insist that a cache revalidate its cache entry with the origin server (and not just with the next cache along the path to the origin server), or to reload its cache entry from the origin server. End-to-end revalidation may be necessary if either the cache or the origin server has overestimated the expiration time of the cached response. End-to-end reload may be necessary if the cache entry has become corrupted for some reason.

End-to-end revalidation may be requested either when the client does not have its own local cached copy, in which case we call it "unspecified end-to-end revalidation", or when the client does have a local cached copy, in which case we call it "specific end-to-end revalidation."

The client can specify these three kinds of action using Cache-Control request directives:

End-to-end reload

The request includes a "no-cache" Cache-Control directive or, for compatibility with HTTP/1.0 clients, "Pragma: no-cache". No field names may be included with the no-cache directive in a request. The server **MUST NOT** use a cached copy when responding to such a request.

Specific end-to-end revalidation

The request includes a "max-age=0" Cache-Control directive, which forces each cache along the path to the origin server to revalidate its own entry, if any, with the next cache or server. The initial

request includes a cache-validating conditional with the client's current validator.

Unspecified end-to-end revalidation

The request includes "max-age=0" Cache-Control directive, which forces each cache along the path to the origin server to revalidate its own entry, if any, with the next cache or server. The initial request does not include a cache-validating conditional; the first cache along the path (if any) that holds a cache entry for this resource includes a cache-validating conditional with its current validator.

When an intermediate cache is forced, by means of a max-age=0 directive, to revalidate its own cache entry, and the client has supplied its own validator in the request, the supplied validator may differ from the validator currently stored with the cache entry. In this case, the cache may use either validator in making its own request without affecting semantic transparency.

However, the choice of validator may affect performance. The best approach is for the intermediate cache to use its own validator when making its request. If the server replies with 304 (Not Modified), then the cache should return its now validated copy to the client with a 200 (OK) response. If the server replies with a new entity and cache validator, however, the intermediate cache should compare the returned validator with the one provided in the client's request, using the strong comparison function. If the client's validator is equal to the origin server's, then the intermediate cache simply returns 304 (Not Modified). Otherwise, it returns the new entity with a 200 (OK) response.

If a request includes the no-cache directive, it should not include min-fresh, max-stale, or max-age.

In some cases, such as times of extremely poor network connectivity, a client may want a cache to return only those responses that it currently has stored, and not to reload or revalidate with the origin server. To do this, the client may include the only-if-cached directive in a request. If it receives this directive, a cache SHOULD either respond using a cached entry that is consistent with the other constraints of the request, or respond with a 504 (Gateway Timeout) status. However, if a group of caches is being operated as a unified system with good internal connectivity, such a request MAY be forwarded within that group of caches.

Because a cache may be configured to ignore a server's specified expiration time, and because a client request may include a max-stale directive (which has a similar effect), the protocol also includes a

mechanism for the origin server to require revalidation of a cache entry on any subsequent use. When the must-revalidate directive is present in a response received by a cache, that cache **MUST NOT** use the entry after it becomes stale to respond to a subsequent request without first revalidating it with the origin server. (I.e., the cache must do an end-to-end revalidation every time, if, based solely on the origin server's Expires or max-age value, the cached response is stale.)

The must-revalidate directive is necessary to support reliable operation for certain protocol features. In all circumstances an HTTP/1.1 cache **MUST** obey the must-revalidate directive; in particular, if the cache cannot reach the origin server for any reason, it **MUST** generate a 504 (Gateway Timeout) response.

Servers should send the must-revalidate directive if and only if failure to revalidate a request on the entity could result in incorrect operation, such as a silently unexecuted financial transaction. Recipients **MUST NOT** take any automated action that violates this directive, and **MUST NOT** automatically provide an unvalidated copy of the entity if revalidation fails.

Although this is not recommended, user agents operating under severe connectivity constraints may violate this directive but, if so, **MUST** explicitly warn the user that an unvalidated response has been provided. The warning **MUST** be provided on each unvalidated access, and **SHOULD** require explicit user confirmation.

The proxy-revalidate directive has the same meaning as the must-revalidate directive, except that it does not apply to non-shared user agent caches. It can be used on a response to an authenticated request to permit the user's cache to store and later return the response without needing to revalidate it (since it has already been authenticated once by that user), while still requiring proxies that service many users to revalidate each time (in order to make sure that each user has been authenticated). Note that such authenticated responses also need the public cache control directive in order to allow them to be cached at all.

14.9.5 No-Transform Directive

Implementers of intermediate caches (proxies) have found it useful to convert the media type of certain entity bodies. A proxy might, for example, convert between image formats in order to save cache space or to reduce the amount of traffic on a slow link. HTTP has to date been silent on these transformations.

Serious operational problems have already occurred, however, when these transformations have been applied to entity bodies intended for certain kinds of applications. For example, applications for medical imaging, scientific data analysis and those using end-to-end authentication, all depend on receiving an entity body that is bit for bit identical to the original entity-body.

Therefore, if a response includes the no-transform directive, an intermediate cache or proxy **MUST NOT** change those headers that are listed in section 13.5.2 as being subject to the no-transform directive. This implies that the cache or proxy must not change any aspect of the entity-body that is specified by these headers.

14.9.6 Cache Control Extensions

The Cache-Control header field can be extended through the use of one or more cache-extension tokens, each with an optional assigned value. Informational extensions (those which do not require a change in cache behavior) may be added without changing the semantics of other directives. Behavioral extensions are designed to work by acting as modifiers to the existing base of cache directives. Both the new directive and the standard directive are supplied, such that applications which do not understand the new directive will default to the behavior specified by the standard directive, and those that understand the new directive will recognize it as modifying the requirements associated with the standard directive. In this way, extensions to the Cache-Control directives can be made without requiring changes to the base protocol.

This extension mechanism depends on a HTTP cache obeying all of the cache-control directives defined for its native HTTP-version, obeying certain extensions, and ignoring all directives that it does not understand.

For example, consider a hypothetical new response directive called "community" which acts as a modifier to the "private" directive. We define this new directive to mean that, in addition to any non-shared cache, any cache which is shared only by members of the community named within its value may cache the response. An origin server wishing to allow the "UCI" community to use an otherwise private response in their shared cache(s) may do so by including

Cache-Control: private, community="UCI"

A cache seeing this header field will act correctly even if the cache does not understand the "community" cache-extension, since it will also see and understand the "private" directive and thus default to the safe behavior.

Unrecognized cache-directives MUST be ignored; it is assumed that any cache-directive likely to be unrecognized by an HTTP/1.1 cache will be combined with standard directives (or the response's default cachability) such that the cache behavior will remain minimally correct even if the cache does not understand the extension(s).

14.10 Connection

The Connection general-header field allows the sender to specify options that are desired for that particular connection and MUST NOT be communicated by proxies over further connections.

The Connection header has the following grammar:

```
Connection-header = "Connection" ":" 1#(connection-token)
connection-token  = token
```

HTTP/1.1 proxies MUST parse the Connection header field before a message is forwarded and, for each connection-token in this field, remove any header field(s) from the message with the same name as the connection-token. Connection options are signaled by the presence of a connection-token in the Connection header field, not by any corresponding additional header field(s), since the additional header field may not be sent if there are no parameters associated with that connection option. HTTP/1.1 defines the "close" connection option for the sender to signal that the connection will be closed after completion of the response. For example,

```
Connection: close
```

in either the request or the response header fields indicates that the connection should not be considered persistent' (section 8.1) after the current request/response is complete.

HTTP/1.1 applications that do not support persistent connections MUST include the "close" connection option in every message.

14.11 Content-Base

The Content-Base entity-header field may be used to specify the base URI for resolving relative URLs within the entity. This header field is described as Base in RFC 1808, which is expected to be revised.

```
Content-Base      = "Content-Base" ":" absoluteURI
```

If no Content-Base field is present, the base URI of an entity is defined either by its Content-Location (if that Content-Location URI is an absolute URI) or the URI used to initiate the request, in that

order of precedence. Note, however, that the base URI of the contents within the entity-body may be redefined within that entity-body.

14.12 Content-Encoding

The Content-Encoding entity-header field is used as a modifier to the media-type. When present, its value indicates what additional content codings have been applied to the entity-body, and thus what decoding mechanisms MUST be applied in order to obtain the media-type referenced by the Content-Type header field. Content-Encoding is primarily used to allow a document to be compressed without losing the identity of its underlying media type.

Content-Encoding = "Content-Encoding" ":" 1#content-coding

Content codings are defined in section 3.5. An example of its use is

Content-Encoding: gzip

The Content-Encoding is a characteristic of the entity identified by the Request-URI. Typically, the entity-body is stored with this encoding and is only decoded before rendering or analogous usage.

If multiple encodings have been applied to an entity, the content codings MUST be listed in the order in which they were applied.

Additional information about the encoding parameters MAY be provided by other entity-header fields not defined by this specification.

14.13 Content-Language

The Content-Language entity-header field describes the natural language(s) of the intended audience for the enclosed entity. Note that this may not be equivalent to all the languages used within the entity-body.

Content-Language = "Content-Language" ":" 1#language-tag

Language tags are defined in section 3.10. The primary purpose of Content-Language is to allow a user to identify and differentiate entities according to the user's own preferred language. Thus, if the body content is intended only for a Danish-literate audience, the appropriate field is

Content-Language: da

If no Content-Language is specified, the default is that the content is intended for all language audiences. This may mean that the sender

does not consider it to be specific to any natural language, or that the sender does not know for which language it is intended.

Multiple languages MAY be listed for content that is intended for multiple audiences. For example, a rendition of the "Treaty of Waitangi," presented simultaneously in the original Maori and English versions, would call for

Content-Language: mi, en

However, just because multiple languages are present within an entity does not mean that it is intended for multiple linguistic audiences. An example would be a beginner's language primer, such as "A First Lesson in Latin," which is clearly intended to be used by an English-literate audience. In this case, the Content-Language should only include "en".

Content-Language may be applied to any media type -- it is not limited to textual documents.

14.14 Content-Length

The Content-Length entity-header field indicates the size of the message-body, in decimal number of octets, sent to the recipient or, in the case of the HEAD method, the size of the entity-body that would have been sent had the request been a GET.

Content-Length = "Content-Length" ":" 1*DIGIT

An example is

Content-Length: 3495

Applications SHOULD use this field to indicate the size of the message-body to be transferred, regardless of the media type of the entity. It must be possible for the recipient to reliably determine the end of HTTP/1.1 requests containing an entity-body, e.g., because the request has a valid Content-Length field, uses Transfer-Encoding: chunked or a multipart body.

Any Content-Length greater than or equal to zero is a valid value. Section 4.4 describes how to determine the length of a message-body if a Content-Length is not given.

Note: The meaning of this field is significantly different from the corresponding definition in MIME, where it is an optional field used within the "message/external-body" content-type. In HTTP, it SHOULD be sent whenever the message's length can be determined prior to being transferred.

14.15 Content-Location

The Content-Location entity-header field may be used to supply the resource location for the entity enclosed in the message. In the case where a resource has multiple entities associated with it, and those entities actually have separate locations by which they might be individually accessed, the server should provide a Content-Location for the particular variant which is returned. In addition, a server SHOULD provide a Content-Location for the resource corresponding to the response entity.

Content-Location = "Content-Location" ":"
(absoluteURI | relativeURI)

If no Content-Base header field is present, the value of Content-Location also defines the base URL for the entity (see section 14.11).

The Content-Location value is not a replacement for the original requested URI; it is only a statement of the location of the resource corresponding to this particular entity at the time of the request. Future requests MAY use the Content-Location URI if the desire is to identify the source of that particular entity.

A cache cannot assume that an entity with a Content-Location different from the URI used to retrieve it can be used to respond to later requests on that Content-Location URI. However, the Content-Location can be used to differentiate between multiple entities retrieved from a single requested resource, as described in section 13.6.

If the Content-Location is a relative URI, the URI is interpreted relative to any Content-Base URI provided in the response. If no Content-Base is provided, the relative URI is interpreted relative to the Request-URI.

14.16 Content-MD5

The Content-MD5 entity-header field, as defined in RFC 1864 [23], is an MD5 digest of the entity-body for the purpose of providing an end-to-end message integrity check (MIC) of the entity-body. (Note: a MIC is good for detecting accidental modification of the entity-body in transit, but is not proof against malicious attacks.)

Content-MD5 = "Content-MD5" ":" md5-digest

md5-digest = <base64 of 128 bit MD5 digest as per RFC 1864>

The Content-MD5 header field may be generated by an origin server to function as an integrity check of the entity-body. Only origin servers may generate the Content-MD5 header field; proxies and gateways MUST NOT generate it, as this would defeat its value as an end-to-end integrity check. Any recipient of the entity-body, including gateways and proxies, MAY check that the digest value in this header field matches that of the entity-body as received.

The MD5 digest is computed based on the content of the entity-body, including any Content-Encoding that has been applied, but not including any Transfer-Encoding that may have been applied to the message-body. If the message is received with a Transfer-Encoding, that encoding must be removed prior to checking the Content-MD5 value against the received entity.

This has the result that the digest is computed on the octets of the entity-body exactly as, and in the order that, they would be sent if no Transfer-Encoding were being applied.

HTTP extends RFC 1864 to permit the digest to be computed for MIME composite media-types (e.g., multipart/* and message/rfc822), but this does not change how the digest is computed as defined in the preceding paragraph.

Note: There are several consequences of this. The entity-body for composite types may contain many body-parts, each with its own MIME and HTTP headers (including Content-MD5, Content-Transfer-Encoding, and Content-Encoding headers). If a body-part has a Content-Transfer-Encoding or Content-Encoding header, it is assumed that the content of the body-part has had the encoding applied, and the body-part is included in the Content-MD5 digest as is -- i.e., after the application. The Transfer-Encoding header field is not allowed within body-parts.

Note: while the definition of Content-MD5 is exactly the same for HTTP as in RFC 1864 for MIME entity-bodies, there are several ways

in which the application of Content-MD5 to HTTP entity-bodies differs from its application to MIME entity-bodies. One is that HTTP, unlike MIME, does not use Content-Transfer-Encoding, and does use Transfer-Encoding and Content-Encoding. Another is that HTTP more frequently uses binary content types than MIME, so it is worth noting that, in such cases, the byte order used to compute the digest is the transmission byte order defined for the type. Lastly, HTTP allows transmission of text types with any of several line break conventions and not just the canonical form using CRLF. Conversion of all line breaks to CRLF should not be done before computing or checking the digest: the line break convention used in the text actually transmitted should be left unaltered when computing the digest.

14.17 Content-Range

The Content-Range entity-header is sent with a partial entity-body to specify where in the full entity-body the partial body should be inserted. It also indicates the total size of the full entity-body. When a server returns a partial response to a client, it must describe both the extent of the range covered by the response, and the length of the entire entity-body.

Content-Range = "Content-Range" ":" content-range-spec

content-range-spec = byte-content-range-spec

byte-content-range-spec = bytes-unit SP first-byte-pos "-"
last-byte-pos "/" entity-length

entity-length = 1*DIGIT

Unlike byte-ranges-specifier values, a byte-content-range-spec may only specify one range, and must contain absolute byte positions for both the first and last byte of the range.

A byte-content-range-spec whose last-byte-pos value is less than its first-byte-pos value, or whose entity-length value is less than or equal to its last-byte-pos value, is invalid. The recipient of an invalid byte-content-range-spec MUST ignore it and any content transferred along with it.

Examples of byte-content-range-spec values, assuming that the entity contains a total of 1234 bytes:

- o The first 500 bytes:
bytes 0-499/1234
- o The second 500 bytes:
bytes 500-999/1234
- o All except for the first 500 bytes:
bytes 500-1233/1234
- o The last 500 bytes:
bytes 734-1233/1234

When an HTTP message includes the content of a single range (for example, a response to a request for a single range, or to a request for a set of ranges that overlap without any holes), this content is transmitted with a Content-Range header, and a Content-Length header showing the number of bytes actually transferred. For example,

```
HTTP/1.1 206 Partial content
Date: Wed, 15 Nov 1995 06:25:24 GMT
Last-modified: Wed, 15 Nov 1995 04:58:08 GMT
Content-Range: bytes 21010-47021/47022
Content-Length: 26012
Content-Type: image/gif
```

When an HTTP message includes the content of multiple ranges (for example, a response to a request for multiple non-overlapping ranges), these are transmitted as a multipart MIME message. The multipart MIME content-type used for this purpose is defined in this specification to be "multipart/byteranges". See appendix 19.2 for its definition.

A client that cannot decode a MIME multipart/byteranges message should not ask for multiple byte-ranges in a single request.

When a client requests multiple byte-ranges in one request, the server SHOULD return them in the order that they appeared in the request.

If the server ignores a byte-range-spec because it is invalid, the server should treat the request as if the invalid Range header field

did not exist. (Normally, this means return a 200 response containing the full entity). The reason is that the only time a client will make such an invalid request is when the entity is smaller than the entity retrieved by a prior request.

14.18 Content-Type

The Content-Type entity-header field indicates the media type of the entity-body sent to the recipient or, in the case of the HEAD method, the media type that would have been sent had the request been a GET.

Content-Type = "Content-Type" ":" media-type
Media types are defined in section 3.7. An example of the field is

Content-Type: text/html; charset=ISO-8859-4

Further discussion of methods for identifying the media type of an entity is provided in section 7.2.1.

14.19 Date

The Date general-header field represents the date and time at which the message was originated, having the same semantics as orig-date in RFC 822. The field value is an HTTP-date, as described in section 3.3.1.

Date = "Date" ":" HTTP-date

An example is

Date: Tue, 15 Nov 1994 08:12:31 GMT

If a message is received via direct connection with the user agent (in the case of requests) or the origin server (in the case of responses), then the date can be assumed to be the current date at the receiving end. However, since the date--as it is believed by the origin--is important for evaluating cached responses, origin servers MUST include a Date header field in all responses. Clients SHOULD only send a Date header field in messages that include an entity-body, as in the case of the PUT and POST requests, and even then it is optional. A received message which does not have a Date header field SHOULD be assigned one by the recipient if the message will be cached by that recipient or gatewayed via a protocol which requires a Date.

In theory, the date SHOULD represent the moment just before the entity is generated. In practice, the date can be generated at any time during the message origination without affecting its semantic value.

The format of the Date is an absolute date and time as defined by HTTP-date in section 3.3; it MUST be sent in RFC1123 [8]-date format.

14.20 ETag

The ETag entity-header field defines the entity tag for the associated entity. The headers used with entity tags are described in sections 14.20, 14.25, 14.26 and 14.43. The entity tag may be used for comparison with other entities from the same resource (see section 13.3.2).

ETag = "ETag" ":" entity-tag

Examples:

ETag: "xyzzzy"
ETag: W/"xyzzzy"
ETag: ""

14.21 Expires

The Expires entity-header field gives the date/time after which the response should be considered stale. A stale cache entry may not normally be returned by a cache (either a proxy cache or an user agent cache) unless it is first validated with the origin server (or with an intermediate cache that has a fresh copy of the entity). See section 13.2 for further discussion of the expiration model.

The presence of an Expires field does not imply that the original resource will change or cease to exist at, before, or after that time.

The format is an absolute date and time as defined by HTTP-date in section 3.3; it MUST be in RFC1123-date format:

Expires = "Expires" ":" HTTP-date

An example of its use is

Expires: Thu, 01 Dec 1994 16:00:00 GMT

Note: if a response includes a Cache-Control field with the max-age directive, that directive overrides the Expires field.

HTTP/1.1 clients and caches MUST treat other invalid date formats, especially including the value "0", as in the past (i.e., "already expired").

To mark a response as "already expired," an origin server should use an Expires date that is equal to the Date header value. (See the rules for expiration calculations in section 13.2.4.)

To mark a response as "never expires," an origin server should use an Expires date approximately one year from the time the response is sent. HTTP/1.1 servers should not send Expires dates more than one year in the future.

The presence of an Expires header field with a date value of some time in the future on an response that otherwise would by default be non-cacheable indicates that the response is cachable, unless indicated otherwise by a Cache-Control header field (section 14.9).

14.22 From

The From request-header field, if given, SHOULD contain an Internet e-mail address for the human user who controls the requesting user agent. The address SHOULD be machine-usable, as defined by mailbox in RFC 822 (as updated by RFC 1123):

From = "From" ":" mailbox

An example is:

From: webmaster@w3.org

This header field MAY be used for logging purposes and as a means for identifying the source of invalid or unwanted requests. It SHOULD NOT be used as an insecure form of access protection. The interpretation of this field is that the request is being performed on behalf of the person given, who accepts responsibility for the method performed. In particular, robot agents SHOULD include this header so that the person responsible for running the robot can be contacted if problems occur on the receiving end.

The Internet e-mail address in this field MAY be separate from the Internet host which issued the request. For example, when a request is passed through a proxy the original issuer's address SHOULD be used.

Note: The client SHOULD not send the From header field without the user's approval, as it may conflict with the user's privacy interests or their site's security policy. It is strongly recommended that the user be able to disable, enable, and modify the value of this field at any time prior to a request.

14.23 Host

The Host request-header field specifies the Internet host and port number of the resource being requested, as obtained from the original URL given by the user or referring resource (generally an HTTP URL, as described in section 3.2.2). The Host field value MUST represent the network location of the origin server or gateway given by the original URL. This allows the origin server or gateway to differentiate between internally-ambiguous URLs, such as the root "/" URL of a server for multiple host names on a single IP address.

Host = "Host" ":" host [":" port] ; Section 3.2.2

A "host" without any trailing port information implies the default port for the service requested (e.g., "80" for an HTTP URL). For example, a request on the origin server for <http://www.w3.org/pub/WWW/> MUST include:

```
GET /pub/WWW/ HTTP/1.1
Host: www.w3.org
```

A client MUST include a Host header field in all HTTP/1.1 request messages on the Internet (i.e., on any message corresponding to a request for a URL which includes an Internet host address for the service being requested). If the Host field is not already present, an HTTP/1.1 proxy MUST add a Host field to the request message prior to forwarding it on the Internet. All Internet-based HTTP/1.1 servers MUST respond with a 400 status code to any HTTP/1.1 request message which lacks a Host header field.

See sections 5.2 and 19.5.1 for other requirements relating to Host.

14.24 If-Modified-Since

The If-Modified-Since request-header field is used with the GET method to make it conditional: if the requested variant has not been modified since the time specified in this field, an entity will not

be returned from the server; instead, a 304 (not modified) response will be returned without any message-body.

If-Modified-Since = "If-Modified-Since" ":" HTTP-date

An example of the field is:

If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT

A GET method with an If-Modified-Since header and no Range header requests that the identified entity be transferred only if it has been modified since the date given by the If-Modified-Since header. The algorithm for determining this includes the following cases:

- a) If the request would normally result in anything other than a 200 (OK) status, or if the passed If-Modified-Since date is invalid, the response is exactly the same as for a normal GET. A date which is later than the server's current time is invalid.
- b) If the variant has been modified since the If-Modified-Since date, the response is exactly the same as for a normal GET.
- c) If the variant has not been modified since a valid If-Modified-Since date, the server **MUST** return a 304 (Not Modified) response.

The purpose of this feature is to allow efficient updates of cached information with a minimum amount of transaction overhead.

Note that the Range request-header field modifies the meaning of If-Modified-Since; see section 14.36 for full details.

Note that If-Modified-Since times are interpreted by the server, whose clock may not be synchronized with the client.

Note that if a client uses an arbitrary date in the If-Modified-Since header instead of a date taken from the Last-Modified header for the same request, the client should be aware of the fact that this date is interpreted in the server's understanding of time. The client should consider unsynchronized clocks and rounding problems due to the different encodings of time between the client and server. This includes the possibility of race conditions if the document has changed between the time it was first requested and the If-Modified-Since date of a subsequent request, and the possibility of clock-skew-related problems if the If-Modified-Since date is derived from the client's clock without correction to the server's clock. Corrections for different time bases between client and server are at best approximate due to network latency.

14.25 If-Match

The If-Match request-header field is used with a method to make it conditional. A client that has one or more entities previously obtained from the resource can verify that one of those entities is current by including a list of their associated entity tags in the If-Match header field. The purpose of this feature is to allow efficient updates of cached information with a minimum amount of transaction overhead. It is also used, on updating requests, to prevent inadvertent modification of the wrong version of a resource. As a special case, the value "*" matches any current entity of the resource.

If-Match = "If-Match" ":" ("*" | 1#entity-tag)

If any of the entity tags match the entity tag of the entity that would have been returned in the response to a similar GET request (without the If-Match header) on that resource, or if "*" is given and any current entity exists for that resource, then the server MAY perform the requested method as if the If-Match header field did not exist.

A server MUST use the strong comparison function (see section 3.11) to compare the entity tags in If-Match.

If none of the entity tags match, or if "*" is given and no current entity exists, the server MUST NOT perform the requested method, and MUST return a 412 (Precondition Failed) response. This behavior is most useful when the client wants to prevent an updating method, such as PUT, from modifying a resource that has changed since the client last retrieved it.

If the request would, without the If-Match header field, result in anything other than a 2xx status, then the If-Match header MUST be ignored.

The meaning of "If-Match: *" is that the method SHOULD be performed if the representation selected by the origin server (or by a cache, possibly using the Vary mechanism, see section 14.43) exists, and MUST NOT be performed if the representation does not exist.

A request intended to update a resource (e.g., a PUT) MAY include an If-Match header field to signal that the request method MUST NOT be applied if the entity corresponding to the If-Match value (a single entity tag) is no longer a representation of that resource. This allows the user to indicate that they do not wish the request to be successful if the resource has been changed without their knowledge. Examples:

```
If-Match: "xyzzzy"
If-Match: "xyzzzy", "r2d2xxxx", "c3piozzzz"
If-Match: *
```

14.26 If-None-Match

The If-None-Match request-header field is used with a method to make it conditional. A client that has one or more entities previously obtained from the resource can verify that none of those entities is current by including a list of their associated entity tags in the If-None-Match header field. The purpose of this feature is to allow efficient updates of cached information with a minimum amount of transaction overhead. It is also used, on updating requests, to prevent inadvertent modification of a resource which was not known to exist.

As a special case, the value "*" matches any current entity of the resource.

If-None-Match = "If-None-Match" ":" ("*" | 1#entity-tag)

If any of the entity tags match the entity tag of the entity that would have been returned in the response to a similar GET request (without the If-None-Match header) on that resource, or if "*" is given and any current entity exists for that resource, then the server MUST NOT perform the requested method. Instead, if the request method was GET or HEAD, the server SHOULD respond with a 304 (Not Modified) response, including the cache-related entity-header fields (particularly ETag) of one of the entities that matched. For all other request methods, the server MUST respond with a status of 412 (Precondition Failed).

See section 13.3.3 for rules on how to determine if two entity tags match. The weak comparison function can only be used with GET or HEAD requests.

If none of the entity tags match, or if "*" is given and no current entity exists, then the server MAY perform the requested method as if the If-None-Match header field did not exist.

If the request would, without the If-None-Match header field, result in anything other than a 2xx status, then the If-None-Match header MUST be ignored.

The meaning of "If-None-Match: *" is that the method MUST NOT be performed if the representation selected by the origin server (or by a cache, possibly using the Vary mechanism, see section 14.43) exists, and SHOULD be performed if the representation does not exist. This feature may be useful in preventing races between PUT operations.

Examples:

```
If-None-Match: "xyzyz"
If-None-Match: W/"xyzyz"
If-None-Match: "xyzyz", "r2d2xxxx", "c3piozzzz"
If-None-Match: W/"xyzyz", W/"r2d2xxxx", W/"c3piozzzz"
If-None-Match: *
```

14.27 If-Range

If a client has a partial copy of an entity in its cache, and wishes to have an up-to-date copy of the entire entity in its cache, it could use the Range request-header with a conditional GET (using either or both of If-Unmodified-Since and If-Match.) However, if the condition fails because the entity has been modified, the client would then have to make a second request to obtain the entire current entity-body.

The If-Range header allows a client to "short-circuit" the second request. Informally, its meaning is if the entity is unchanged, send me the part(s) that I am missing; otherwise, send me the entire new entity.

If-Range = "If-Range" ":" (entity-tag | HTTP-date)

If the client has no entity tag for an entity, but does have a Last-Modified date, it may use that date in a If-Range header. (The server can distinguish between a valid HTTP-date and any form of entity-tag by examining no more than two characters.) The If-Range header should only be used together with a Range header, and must be ignored if the request does not include a Range header, or if the server does not support the sub-range operation.

If the entity tag given in the If-Range header matches the current entity tag for the entity, then the server should provide the specified sub-range of the entity using a 206 (Partial content) response. If the entity tag does not match, then the server should return the entire entity using a 200 (OK) response.

14.28 If-Unmodified-Since

The If-Unmodified-Since request-header field is used with a method to make it conditional. If the requested resource has not been modified since the time specified in this field, the server should perform the requested operation as if the If-Unmodified-Since header were not present.

If the requested variant has been modified since the specified time, the server **MUST NOT** perform the requested operation, and **MUST** return a 412 (Precondition Failed).

If-Unmodified-Since = "If-Unmodified-Since" ":" HTTP-date

An example of the field is:

If-Unmodified-Since: Sat, 29 Oct 1994 19:43:31 GMT

If the request normally (i.e., without the If-Unmodified-Since header) would result in anything other than a 2xx status, the If-Unmodified-Since header should be ignored.

If the specified date is invalid, the header is ignored.

14.29 Last-Modified

The Last-Modified entity-header field indicates the date and time at which the origin server believes the variant was last modified.

Last-Modified = "Last-Modified" ":" HTTP-date

An example of its use is

Last-Modified: Tue, 15 Nov 1994 12:45:26 GMT

The exact meaning of this header field depends on the implementation of the origin server and the nature of the original resource. For files, it may be just the file system last-modified time. For entities with dynamically included parts, it may be the most recent of the set of last-modify times for its component parts. For database gateways, it may be the last-update time stamp of the record. For virtual objects, it may be the last time the internal state changed.

An origin server **MUST NOT** send a Last-Modified date which is later than the server's time of message origination. In such cases, where the resource's last modification would indicate some time in the future, the server **MUST** replace that date with the message origination date.

An origin server should obtain the Last-Modified value of the entity as close as possible to the time that it generates the Date value of its response. This allows a recipient to make an accurate assessment of the entity's modification time, especially if the entity changes near the time that the response is generated.

HTTP/1.1 servers **SHOULD** send Last-Modified whenever feasible.

14.30 Location

The Location response-header field is used to redirect the recipient to a location other than the Request-URI for completion of the request or identification of a new resource. For 201 (Created) responses, the Location is that of the new resource which was created by the request. For 3xx responses, the location **SHOULD** indicate the server's preferred URL for automatic redirection to the resource. The field value consists of a single absolute URL.

Location = "Location" ":" absoluteURI

An example is

Location: http://www.w3.org/pub/WWW/People.html

Note: The Content-Location header field (section 14.15) differs from Location in that the Content-Location identifies the original location of the entity enclosed in the request. It is therefore possible for a response to contain header fields for both Location and Content-Location. Also see section 13.10 for cache requirements of some methods.

14.31 Max-Forwards

The Max-Forwards request-header field may be used with the TRACE method (section 14.31) to limit the number of proxies or gateways that can forward the request to the next inbound server. This can be useful when the client is attempting to trace a request chain which appears to be failing or looping in mid-chain.

Max-Forwards = "Max-Forwards" ":" 1*DIGIT

The Max-Forwards value is a decimal integer indicating the remaining number of times this request message may be forwarded.

Each proxy or gateway recipient of a TRACE request containing a Max-Forwards header field SHOULD check and update its value prior to forwarding the request. If the received value is zero (0), the recipient SHOULD NOT forward the request; instead, it SHOULD respond as the final recipient with a 200 (OK) response containing the received request message as the response entity-body (as described in section 9.8). If the received Max-Forwards value is greater than zero, then the forwarded message SHOULD contain an updated Max-Forwards field with a value decremented by one (1).

The Max-Forwards header field SHOULD be ignored for all other methods defined by this specification and for any extension methods for which it is not explicitly referred to as part of that method definition.

14.32 Pragma

The Pragma general-header field is used to include implementation-specific directives that may apply to any recipient along the request/response chain. All pragma directives specify optional behavior from the viewpoint of the protocol; however, some systems MAY require that behavior be consistent with the directives.

```

Pragma           = "Pragma" ":" 1#pragma-directive
pragma-directive = "no-cache" | extension-pragma
extension-pragma = token [ "=" ( token | quoted-string ) ]

```

When the no-cache directive is present in a request message, an application SHOULD forward the request toward the origin server even if it has a cached copy of what is being requested. This pragma directive has the same semantics as the no-cache cache-directive (see section 14.9) and is defined here for backwards compatibility with HTTP/1.0. Clients SHOULD include both header fields when a no-cache request is sent to a server not known to be HTTP/1.1 compliant.

Pragma directives MUST be passed through by a proxy or gateway application, regardless of their significance to that application, since the directives may be applicable to all recipients along the request/response chain. It is not possible to specify a pragma for a specific recipient; however, any pragma directive not relevant to a recipient SHOULD be ignored by that recipient.

HTTP/1.1 clients SHOULD NOT send the Pragma request-header. HTTP/1.1 caches SHOULD treat "Pragma: no-cache" as if the client had sent "Cache-Control: no-cache". No new Pragma directives will be defined in HTTP.

14.33 Proxy-Authenticate

The Proxy-Authenticate response-header field MUST be included as part of a 407 (Proxy Authentication Required) response. The field value consists of a challenge that indicates the authentication scheme and parameters applicable to the proxy for this Request-URI.

Proxy-Authenticate = "Proxy-Authenticate" ":" challenge

The HTTP access authentication process is described in section 11. Unlike WWW-Authenticate, the Proxy-Authenticate header field applies only to the current connection and SHOULD NOT be passed on to downstream clients. However, an intermediate proxy may need to obtain its own credentials by requesting them from the downstream client, which in some circumstances will appear as if the proxy is forwarding the Proxy-Authenticate header field.

14.34 Proxy-Authorization

The Proxy-Authorization request-header field allows the client to identify itself (or its user) to a proxy which requires authentication. The Proxy-Authorization field value consists of credentials containing the authentication information of the user agent for the proxy and/or realm of the resource being requested.

Proxy-Authorization = "Proxy-Authorization" ":" credentials

The HTTP access authentication process is described in section 11. Unlike Authorization, the Proxy-Authorization header field applies only to the next outbound proxy that demanded authentication using the Proxy-Authenticate field. When multiple proxies are used in a chain, the Proxy-Authorization header field is consumed by the first outbound proxy that was expecting to receive credentials. A proxy MAY relay the credentials from the client request to the next proxy if that is the mechanism by which the proxies cooperatively authenticate a given request.

14.35 Public

The Public response-header field lists the set of methods supported by the server. The purpose of this field is strictly to inform the recipient of the capabilities of the server regarding unusual methods. The methods listed may or may not be applicable to the

Request-URI; the Allow header field (section 14.7) MAY be used to indicate methods allowed for a particular URI.

Public = "Public" ":" 1#method

Example of use:

Public: OPTIONS, MGET, MHEAD, GET, HEAD

This header field applies only to the server directly connected to the client (i.e., the nearest neighbor in a chain of connections). If the response passes through a proxy, the proxy MUST either remove the Public header field or replace it with one applicable to its own capabilities.

14.36 Range

14.36.1 Byte Ranges

Since all HTTP entities are represented in HTTP messages as sequences of bytes, the concept of a byte range is meaningful for any HTTP entity. (However, not all clients and servers need to support byte-range operations.)

Byte range specifications in HTTP apply to the sequence of bytes in the entity-body (not necessarily the same as the message-body).

A byte range operation may specify a single range of bytes, or a set of ranges within a single entity.

ranges-specifier = byte-ranges-specifier

byte-ranges-specifier = bytes-unit "=" byte-range-set

byte-range-set = 1#(byte-range-spec | suffix-byte-range-spec)

byte-range-spec = first-byte-pos "-" [last-byte-pos]

first-byte-pos = 1*DIGIT

last-byte-pos = 1*DIGIT

The first-byte-pos value in a byte-range-spec gives the byte-offset of the first byte in a range. The last-byte-pos value gives the byte-offset of the last byte in the range; that is, the byte positions specified are inclusive. Byte offsets start at zero.

If the last-byte-pos value is present, it must be greater than or equal to the first-byte-pos in that byte-range-spec, or the byte-range-spec is invalid. The recipient of an invalid byte-range-spec must ignore it.

If the last-byte-pos value is absent, or if the value is greater than or equal to the current length of the entity-body, last-byte-pos is taken to be equal to one less than the current length of the entity-body in bytes.

By its choice of last-byte-pos, a client can limit the number of bytes retrieved without knowing the size of the entity.

suffix-byte-range-spec = "-" suffix-length

suffix-length = 1*DIGIT

A suffix-byte-range-spec is used to specify the suffix of the entity-body, of a length given by the suffix-length value. (That is, this form specifies the last N bytes of an entity-body.) If the entity is shorter than the specified suffix-length, the entire entity-body is used.

Examples of byte-ranges-specifier values (assuming an entity-body of length 10000):

- o The first 500 bytes (byte offsets 0-499, inclusive):
bytes=0-499
- o The second 500 bytes (byte offsets 500-999, inclusive):
bytes=500-999
- o The final 500 bytes (byte offsets 9500-9999, inclusive):
bytes=-500
- o Or
bytes=9500-
- o The first and last bytes only (bytes 0 and 9999):
bytes=0-0,-1

- o Several legal but not canonical specifications of the second 500 bytes (byte offsets 500-999, inclusive):

bytes=500-600, 601-999

bytes=500-700, 601-999

14.36.2 Range Retrieval Requests

HTTP retrieval requests using conditional or unconditional GET methods may request one or more sub-ranges of the entity, instead of the entire entity, using the Range request header, which applies to the entity returned as the result of the request:

Range = "Range" ":" ranges-specifier

A server MAY ignore the Range header. However, HTTP/1.1 origin servers and intermediate caches SHOULD support byte ranges when possible, since Range supports efficient recovery from partially failed transfers, and supports efficient partial retrieval of large entities.

If the server supports the Range header and the specified range or ranges are appropriate for the entity:

- o The presence of a Range header in an unconditional GET modifies what is returned if the GET is otherwise successful. In other words, the response carries a status code of 206 (Partial Content) instead of 200 (OK).
- o The presence of a Range header in a conditional GET (a request using one or both of If-Modified-Since and If-None-Match, or one or both of If-Unmodified-Since and If-Match) modifies what is returned if the GET is otherwise successful and the condition is true. It does not affect the 304 (Not Modified) response returned if the conditional is false.

In some cases, it may be more appropriate to use the If-Range header (see section 14.27) in addition to the Range header.

If a proxy that supports ranges receives a Range request, forwards the request to an inbound server, and receives an entire entity in reply, it SHOULD only return the requested range to its client. It SHOULD store the entire received response in its cache, if that is consistent with its cache allocation policies.

14.37 Referer

The Referer[sic] request-header field allows the client to specify, for the server's benefit, the address (URI) of the resource from which the Request-URI was obtained (the "referrer", although the header field is misspelled.) The Referer request-header allows a server to generate lists of back-links to resources for interest, logging, optimized caching, etc. It also allows obsolete or mistyped links to be traced for maintenance. The Referer field MUST NOT be sent if the Request-URI was obtained from a source that does not have its own URI, such as input from the user keyboard.

Referer = "Referer" ":" (absoluteURI | relativeURI)

Example:

Referer: http://www.w3.org/hypertext/DataSources/Overview.html

If the field value is a partial URI, it SHOULD be interpreted relative to the Request-URI. The URI MUST NOT include a fragment.

Note: Because the source of a link may be private information or may reveal an otherwise private information source, it is strongly recommended that the user be able to select whether or not the Referer field is sent. For example, a browser client could have a toggle switch for browsing openly/anonymously, which would respectively enable/disable the sending of Referer and From information.

14.38 Retry-After

The Retry-After response-header field can be used with a 503 (Service Unavailable) response to indicate how long the service is expected to be unavailable to the requesting client. The value of this field can be either an HTTP-date or an integer number of seconds (in decimal) after the time of the response.

Retry-After = "Retry-After" ":" (HTTP-date | delta-seconds)

Two examples of its use are

Retry-After: Fri, 31 Dec 1999 23:59:59 GMT
Retry-After: 120

In the latter example, the delay is 2 minutes.

14.39 Server

The Server response-header field contains information about the software used by the origin server to handle the request. The field can contain multiple product tokens (section 3.8) and comments identifying the server and any significant subproducts. The product tokens are listed in order of their significance for identifying the application.

Server = "Server" ":" 1*(product | comment)

Example:

Server: CERN/3.0 libwww/2.17

If the response is being forwarded through a proxy, the proxy application MUST NOT modify the Server response-header. Instead, it SHOULD include a Via field (as described in section 14.44).

Note: Revealing the specific software version of the server may allow the server machine to become more vulnerable to attacks against software that is known to contain security holes. Server implementers are encouraged to make this field a configurable option.

14.40 Transfer-Encoding

The Transfer-Encoding general-header field indicates what (if any) type of transformation has been applied to the message body in order to safely transfer it between the sender and the recipient. This differs from the Content-Encoding in that the transfer coding is a property of the message, not of the entity.

Transfer-Encoding = "Transfer-Encoding" ":" 1#transfer-coding

Transfer codings are defined in section 3.6. An example is:

Transfer-Encoding: chunked

Many older HTTP/1.0 applications do not understand the Transfer-Encoding header.

14.41 Upgrade

The Upgrade general-header allows the client to specify what additional communication protocols it supports and would like to use if the server finds it appropriate to switch protocols. The server

MUST use the Upgrade header field within a 101 (Switching Protocols) response to indicate which protocol(s) are being switched.

Upgrade = "Upgrade" ":" 1#product

For example,

Upgrade: HTTP/2.0, SHHTTP/1.3, IRC/6.9, RTA/x11

The Upgrade header field is intended to provide a simple mechanism for transition from HTTP/1.1 to some other, incompatible protocol. It does so by allowing the client to advertise its desire to use another protocol, such as a later version of HTTP with a higher major version number, even though the current request has been made using HTTP/1.1. This eases the difficult transition between incompatible protocols by allowing the client to initiate a request in the more commonly supported protocol while indicating to the server that it would like to use a "better" protocol if available (where "better" is determined by the server, possibly according to the nature of the method and/or resource being requested).

The Upgrade header field only applies to switching application-layer protocols upon the existing transport-layer connection. Upgrade cannot be used to insist on a protocol change; its acceptance and use by the server is optional. The capabilities and nature of the application-layer communication after the protocol change is entirely dependent upon the new protocol chosen, although the first action after changing the protocol MUST be a response to the initial HTTP request containing the Upgrade header field.

The Upgrade header field only applies to the immediate connection. Therefore, the upgrade keyword MUST be supplied within a Connection header field (section 14.10) whenever Upgrade is present in an HTTP/1.1 message.

The Upgrade header field cannot be used to indicate a switch to a protocol on a different connection. For that purpose, it is more appropriate to use a 301, 302, 303, or 305 redirection response.

This specification only defines the protocol name "HTTP" for use by the family of Hypertext Transfer Protocols, as defined by the HTTP version rules of section 3.1 and future updates to this specification. Any token can be used as a protocol name; however, it will only be useful if both the client and server associate the name with the same protocol.

14.42 User-Agent

The User-Agent request-header field contains information about the user agent originating the request. This is for statistical purposes, the tracing of protocol violations, and automated recognition of user agents for the sake of tailoring responses to avoid particular user agent limitations. User agents SHOULD include this field with requests. The field can contain multiple product tokens (section 3.8) and comments identifying the agent and any subproducts which form a significant part of the user agent. By convention, the product tokens are listed in order of their significance for identifying the application.

User-Agent = "User-Agent" ":" 1*(product | comment)

Example:

User-Agent: CERN-LineMode/2.15 libwww/2.17b3

14.43 Vary

The Vary response-header field is used by a server to signal that the response entity was selected from the available representations of the response using server-driven negotiation (section 12). Field-names listed in Vary headers are those of request-headers. The Vary field value indicates either that the given set of header fields encompass the dimensions over which the representation might vary, or that the dimensions of variance are unspecified ("*") and thus may vary over any aspect of future requests.

Vary = "Vary" ":" ("*" | 1#field-name)

An HTTP/1.1 server MUST include an appropriate Vary header field with any cachable response that is subject to server-driven negotiation. Doing so allows a cache to properly interpret future requests on that resource and informs the user agent about the presence of negotiation on that resource. A server SHOULD include an appropriate Vary header field with a non-cachable response that is subject to server-driven negotiation, since this might provide the user agent with useful information about the dimensions over which the response might vary.

The set of header fields named by the Vary field value is known as the "selecting" request-headers.

When the cache receives a subsequent request whose Request-URI specifies one or more cache entries including a Vary header, the cache MUST NOT use such a cache entry to construct a response to the new request unless all of the headers named in the cached Vary header

are present in the new request, and all of the stored selecting request-headers from the previous request match the corresponding headers in the new request.

The selecting request-headers from two requests are defined to match if and only if the selecting request-headers in the first request can be transformed to the selecting request-headers in the second request by adding or removing linear whitespace (LWS) at places where this is allowed by the corresponding BNF, and/or combining multiple message-header fields with the same field name following the rules about message headers in section 4.2.

A Vary field value of "*" signals that unspecified parameters, possibly other than the contents of request-header fields (e.g., the network address of the client), play a role in the selection of the response representation. Subsequent requests on that resource can only be properly interpreted by the origin server, and thus a cache **MUST** forward a (possibly conditional) request even when it has a fresh response cached for the resource. See section 13.6 for use of the Vary header by caches.

A Vary field value consisting of a list of field-names signals that the representation selected for the response is based on a selection algorithm which considers **ONLY** the listed request-header field values in selecting the most appropriate representation. A cache **MAY** assume that the same selection will be made for future requests with the same values for the listed field names, for the duration of time in which the response is fresh.

The field-names given are not limited to the set of standard request-header fields defined by this specification. Field names are case-insensitive.

14.44 Via

The Via general-header field **MUST** be used by gateways and proxies to indicate the intermediate protocols and recipients between the user agent and the server on requests, and between the origin server and the client on responses. It is analogous to the "Received" field of RFC 822 and is intended to be used for tracking message forwards, avoiding request loops, and identifying the protocol capabilities of all senders along the request/response chain.

```

Via = "Via" ":" 1#( received-protocol received-by [ comment ] )

received-protocol = [ protocol-name "/" ] protocol-version
protocol-name     = token
protocol-version  = token
received-by       = ( host [ ":" port ] ) | pseudonym
pseudonym         = token

```

The received-protocol indicates the protocol version of the message received by the server or client along each segment of the request/response chain. The received-protocol version is appended to the Via field value when the message is forwarded so that information about the protocol capabilities of upstream applications remains visible to all recipients.

The protocol-name is optional if and only if it would be "HTTP". The received-by field is normally the host and optional port number of a recipient server or client that subsequently forwarded the message. However, if the real host is considered to be sensitive information, it MAY be replaced by a pseudonym. If the port is not given, it MAY be assumed to be the default port of the received-protocol.

Multiple Via field values represent each proxy or gateway that has forwarded the message. Each recipient MUST append its information such that the end result is ordered according to the sequence of forwarding applications.

Comments MAY be used in the Via header field to identify the software of the recipient proxy or gateway, analogous to the User-Agent and Server header fields. However, all comments in the Via field are optional and MAY be removed by any recipient prior to forwarding the message.

For example, a request message could be sent from an HTTP/1.0 user agent to an internal proxy code-named "fred", which uses HTTP/1.1 to forward the request to a public proxy at nowhere.com, which completes the request by forwarding it to the origin server at www.ics.uci.edu. The request received by www.ics.uci.edu would then have the following Via header field:

```
Via: 1.0 fred, 1.1 nowhere.com (Apache/1.1)
```

Proxies and gateways used as a portal through a network firewall SHOULD NOT, by default, forward the names and ports of hosts within the firewall region. This information SHOULD only be propagated if explicitly enabled. If not enabled, the received-by host of any host behind the firewall SHOULD be replaced by an appropriate pseudonym for that host.

For organizations that have strong privacy requirements for hiding internal structures, a proxy MAY combine an ordered subsequence of Via header field entries with identical received-protocol values into a single such entry. For example,

Via: 1.0 ricky, 1.1 ethel, 1.1 fred, 1.0 lucy

could be collapsed to

Via: 1.0 ricky, 1.1 mertz, 1.0 lucy

Applications SHOULD NOT combine multiple entries unless they are all under the same organizational control and the hosts have already been replaced by pseudonyms. Applications MUST NOT combine entries which have different received-protocol values.

14.45 Warning

The Warning response-header field is used to carry additional information about the status of a response which may not be reflected by the response status code. This information is typically, though not exclusively, used to warn about a possible lack of semantic transparency from caching operations.

Warning headers are sent with responses using:

Warning = "Warning" ":" 1#warning-value

warning-value = warn-code SP warn-agent SP warn-text

warn-code = 2DIGIT

warn-agent = (host [":" port]) | pseudonym

; the name or pseudonym of the server adding

; the Warning header, for use in debugging

warn-text = quoted-string

A response may carry more than one Warning header.

The warn-text should be in a natural language and character set that is most likely to be intelligible to the human user receiving the response. This decision may be based on any available knowledge, such as the location of the cache or user, the Accept-Language field in a request, the Content-Language field in a response, etc. The default language is English and the default character set is ISO-8859-1.

If a character set other than ISO-8859-1 is used, it MUST be encoded in the warn-text using the method described in RFC 1522 [14].

Any server or cache may add Warning headers to a response. New Warning headers should be added after any existing Warning headers. A cache MUST NOT delete any Warning header that it received with a response. However, if a cache successfully validates a cache entry, it SHOULD remove any Warning headers previously attached to that entry except as specified for specific Warning codes. It MUST then add any Warning headers received in the validating response. In other words, Warning headers are those that would be attached to the most recent relevant response.

When multiple Warning headers are attached to a response, the user agent SHOULD display as many of them as possible, in the order that they appear in the response. If it is not possible to display all of the warnings, the user agent should follow these heuristics:

- o Warnings that appear early in the response take priority over those appearing later in the response.
- o Warnings in the user's preferred character set take priority over warnings in other character sets but with identical warn-codes and warn-agents.

Systems that generate multiple Warning headers should order them with this user agent behavior in mind.

This is a list of the currently-defined warn-codes, each with a recommended warn-text in English, and a description of its meaning.

10 Response is stale

MUST be included whenever the returned response is stale. A cache may add this warning to any response, but may never remove it until the response is known to be fresh.

11 Revalidation failed

MUST be included if a cache returns a stale response because an attempt to revalidate the response failed, due to an inability to reach the server. A cache may add this warning to any response, but may never remove it until the response is successfully revalidated.

12 Disconnected operation

SHOULD be included if the cache is intentionally disconnected from the rest of the network for a period of time.

13 Heuristic expiration

MUST be included if the cache heuristically chose a freshness lifetime greater than 24 hours and the response's age is greater than 24 hours.

14 Transformation applied

MUST be added by an intermediate cache or proxy if it applies any transformation changing the content-coding (as specified in the Content-Encoding header) or media-type (as specified in the Content-Type header) of the response, unless this Warning code already appears in the response. MUST NOT be deleted from a response even after revalidation.

99 Miscellaneous warning

The warning text may include arbitrary information to be presented to a human user, or logged. A system receiving this warning MUST NOT take any automated action.

14.46 WWW-Authenticate

The WWW-Authenticate response-header field MUST be included in 401 (Unauthorized) response messages. The field value consists of at least one challenge that indicates the authentication scheme(s) and parameters applicable to the Request-URI.

WWW-Authenticate = "WWW-Authenticate" ":" 1#challenge

The HTTP access authentication process is described in section 11. User agents MUST take special care in parsing the WWW-Authenticate field value if it contains more than one challenge, or if more than one WWW-Authenticate header field is provided, since the contents of a challenge may itself contain a comma-separated list of authentication parameters.

15 Security Considerations

This section is meant to inform application developers, information providers, and users of the security limitations in HTTP/1.1 as described by this document. The discussion does not include definitive solutions to the problems revealed, though it does make some suggestions for reducing security risks.

15.1 Authentication of Clients

The Basic authentication scheme is not a secure method of user authentication, nor does it in any way protect the entity, which is transmitted in clear text across the physical network used as the carrier. HTTP does not prevent additional authentication schemes and encryption mechanisms from being employed to increase security or the addition of enhancements (such as schemes to use one-time passwords) to Basic authentication.

The most serious flaw in Basic authentication is that it results in the essentially clear text transmission of the user's password over the physical network. It is this problem which Digest Authentication attempts to address.

Because Basic authentication involves the clear text transmission of passwords it **SHOULD** never be used (without enhancements) to protect sensitive or valuable information.

A common use of Basic authentication is for identification purposes -- requiring the user to provide a user name and password as a means of identification, for example, for purposes of gathering accurate usage statistics on a server. When used in this way it is tempting to think that there is no danger in its use if illicit access to the protected documents is not a major concern. This is only correct if the server issues both user name and password to the users and in particular does not allow the user to choose his or her own password. The danger arises because naive users frequently reuse a single password to avoid the task of maintaining multiple passwords.

If a server permits users to select their own passwords, then the threat is not only illicit access to documents on the server but also illicit access to the accounts of all users who have chosen to use their account password. If users are allowed to choose their own password that also means the server must maintain files containing the (presumably encrypted) passwords. Many of these may be the account passwords of users perhaps at distant sites. The owner or administrator of such a system could conceivably incur liability if this information is not maintained in a secure fashion.

Basic Authentication is also vulnerable to spoofing by counterfeit servers. If a user can be led to believe that he is connecting to a host containing information protected by basic authentication when in fact he is connecting to a hostile server or gateway then the attacker can request a password, store it for later use, and feign an error. This type of attack is not possible with Digest Authentication [32]. Server implementers **SHOULD** guard against the possibility of this sort of counterfeiting by gateways or CGI scripts. In particular it is very dangerous for a server to simply turn over a connection to a gateway since that gateway can then use the persistent connection mechanism to engage in multiple transactions with the client while impersonating the original server in a way that is not detectable by the client.

15.2 Offering a Choice of Authentication Schemes

An HTTP/1.1 server may return multiple challenges with a 401 (Authenticate) response, and each challenge may use a different

scheme. The order of the challenges returned to the user agent is in the order that the server would prefer they be chosen. The server should order its challenges with the "most secure" authentication scheme first. A user agent should choose as the challenge to be made to the user the first one that the user agent understands.

When the server offers choices of authentication schemes using the WWW-Authenticate header, the "security" of the authentication is only as malicious user could capture the set of challenges and try to authenticate him/herself using the weakest of the authentication schemes. Thus, the ordering serves more to protect the user's credentials than the server's information.

A possible man-in-the-middle (MITM) attack would be to add a weak authentication scheme to the set of choices, hoping that the client will use one that exposes the user's credentials (e.g. password). For this reason, the client should always use the strongest scheme that it understands from the choices accepted.

An even better MITM attack would be to remove all offered choices, and to insert a challenge that requests Basic authentication. For this reason, user agents that are concerned about this kind of attack could remember the strongest authentication scheme ever requested by a server and produce a warning message that requires user confirmation before using a weaker one. A particularly insidious way to mount such a MITM attack would be to offer a "free" proxy caching service to gullible users.

15.3 Abuse of Server Log Information

A server is in the position to save personal data about a user's requests which may identify their reading patterns or subjects of interest. This information is clearly confidential in nature and its handling may be constrained by law in certain countries. People using the HTTP protocol to provide data are responsible for ensuring that such material is not distributed without the permission of any individuals that are identifiable by the published results.

15.4 Transfer of Sensitive Information

Like any generic data transfer protocol, HTTP cannot regulate the content of the data that is transferred, nor is there any a priori method of determining the sensitivity of any particular piece of information within the context of any given request. Therefore, applications SHOULD supply as much control over this information as possible to the provider of that information. Four header fields are worth special mention in this context: Server, Via, Referer and From.

Revealing the specific software version of the server may allow the server machine to become more vulnerable to attacks against software that is known to contain security holes. Implementers SHOULD make the Server header field a configurable option.

Proxies which serve as a portal through a network firewall SHOULD take special precautions regarding the transfer of header information that identifies the hosts behind the firewall. In particular, they SHOULD remove, or replace with sanitized versions, any Via fields generated behind the firewall.

The Referer field allows reading patterns to be studied and reverse links drawn. Although it can be very useful, its power can be abused if user details are not separated from the information contained in the Referer. Even when the personal information has been removed, the Referer field may indicate a private document's URI whose publication would be inappropriate.

The information sent in the From field might conflict with the user's privacy interests or their site's security policy, and hence it SHOULD NOT be transmitted without the user being able to disable, enable, and modify the contents of the field. The user MUST be able to set the contents of this field within a user preference or application defaults configuration.

We suggest, though do not require, that a convenient toggle interface be provided for the user to enable or disable the sending of From and Referer information.

15.5 Attacks Based On File and Path Names

Implementations of HTTP origin servers SHOULD be careful to restrict the documents returned by HTTP requests to be only those that were intended by the server administrators. If an HTTP server translates HTTP URIs directly into file system calls, the server MUST take special care not to serve files that were not intended to be delivered to HTTP clients. For example, UNIX, Microsoft Windows, and other operating systems use ".." as a path component to indicate a directory level above the current one. On such a system, an HTTP server MUST disallow any such construct in the Request-URI if it would otherwise allow access to a resource outside those intended to be accessible via the HTTP server. Similarly, files intended for reference only internally to the server (such as access control files, configuration files, and script code) MUST be protected from inappropriate retrieval, since they might contain sensitive information. Experience has shown that minor bugs in such HTTP server implementations have turned into security risks.

15.6 Personal Information

HTTP clients are often privy to large amounts of personal information (e.g. the user's name, location, mail address, passwords, encryption keys, etc.), and SHOULD be very careful to prevent unintentional leakage of this information via the HTTP protocol to other sources. We very strongly recommend that a convenient interface be provided for the user to control dissemination of such information, and that designers and implementers be particularly careful in this area. History shows that errors in this area are often both serious security and/or privacy problems, and often generate highly adverse publicity for the implementer's company.

15.7 Privacy Issues Connected to Accept Headers

Accept request-headers can reveal information about the user to all servers which are accessed. The Accept-Language header in particular can reveal information the user would consider to be of a private nature, because the understanding of particular languages is often strongly correlated to the membership of a particular ethnic group. User agents which offer the option to configure the contents of an Accept-Language header to be sent in every request are strongly encouraged to let the configuration process include a message which makes the user aware of the loss of privacy involved.

An approach that limits the loss of privacy would be for a user agent to omit the sending of Accept-Language headers by default, and to ask the user whether it should start sending Accept-Language headers to a server if it detects, by looking for any Vary response-header fields generated by the server, that such sending could improve the quality of service.

Elaborate user-customized accept header fields sent in every request, in particular if these include quality values, can be used by servers as relatively reliable and long-lived user identifiers. Such user identifiers would allow content providers to do click-trail tracking, and would allow collaborating content providers to match cross-server click-trails or form submissions of individual users. Note that for many users not behind a proxy, the network address of the host running the user agent will also serve as a long-lived user identifier. In environments where proxies are used to enhance privacy, user agents should be conservative in offering accept header configuration options to end users. As an extreme privacy measure, proxies could filter the accept headers in relayed requests. General purpose user agents which provide a high degree of header configurability should warn users about the loss of privacy which can be involved.

15.8 DNS Spoofing

Clients using HTTP rely heavily on the Domain Name Service, and are thus generally prone to security attacks based on the deliberate mis-association of IP addresses and DNS names. Clients need to be cautious in assuming the continuing validity of an IP number/DNS name association.

In particular, HTTP clients SHOULD rely on their name resolver for confirmation of an IP number/DNS name association, rather than caching the result of previous host name lookups. Many platforms already can cache host name lookups locally when appropriate, and they SHOULD be configured to do so. These lookups should be cached, however, only when the TTL (Time To Live) information reported by the name server makes it likely that the cached information will remain useful.

If HTTP clients cache the results of host name lookups in order to achieve a performance improvement, they MUST observe the TTL information reported by DNS.

If HTTP clients do not observe this rule, they could be spoofed when a previously-accessed server's IP address changes. As network renumbering is expected to become increasingly common, the possibility of this form of attack will grow. Observing this requirement thus reduces this potential security vulnerability.

This requirement also improves the load-balancing behavior of clients for replicated servers using the same DNS name and reduces the likelihood of a user's experiencing failure in accessing sites which use that strategy.

15.9 Location Headers and Spoofing

If a single server supports multiple organizations that do not trust one another, then it must check the values of Location and Content-Location headers in responses that are generated under control of said organizations to make sure that they do not attempt to invalidate resources over which they have no authority.

16 Acknowledgments

This specification makes heavy use of the augmented BNF and generic constructs defined by David H. Crocker for RFC 822. Similarly, it reuses many of the definitions provided by Nathaniel Borenstein and Ned Freed for MIME. We hope that their inclusion in this specification will help reduce past confusion over the relationship between HTTP and Internet mail message formats.

The HTTP protocol has evolved considerably over the past four years. It has benefited from a large and active developer community--the many people who have participated on the www-talk mailing list--and it is that community which has been most responsible for the success of HTTP and of the World-Wide Web in general. Marc Andreessen, Robert Cailliau, Daniel W. Connolly, Bob Denny, John Franks, Jean-Francois Groff, Phillip M. Hallam-Baker, Hakon W. Lie, Ari Luotonen, Rob McCool, Lou Montulli, Dave Raggett, Tony Sanders, and Marc VanHeyningen deserve special recognition for their efforts in defining early aspects of the protocol.

This document has benefited greatly from the comments of all those participating in the HTTP-WG. In addition to those already mentioned, the following individuals have contributed to this specification:

Gary Adams	Albert Lunde
Harald Tveit Alvestrand	John C. Mallery
Keith Ball	Jean-Philippe Martin-Flatin
Brian Behlendorf	Larry Masinter
Paul Burchard	Mitra
Maurizio Codogno	David Morris
Mike Cowlishaw	Gavin Nicol
Roman Czyborra	Bill Perry
Michael A. Dolan	Jeffrey Perry
David J. Fiander	Scott Powers
Alan Freier	Owen Rees
Marc Hedlund	Luigi Rizzo
Greg Herlihy	David Robinson
Koen Holtman	Marc Salomon
Alex Hopmann	Rich Salz
Bob Jernigan	Allan M. Schiffman
Shel Kaphan	Jim Seidman
Rohit Khare	Chuck Shotton
John Klensin	Eric W. Sink
Martijn Koster	Simon E. Spero
Alexei Kosut	Richard N. Taylor
David M. Kristol	Robert S. Thau
Daniel LaLiberte	Bill (BearHeart) Weinman
Ben Laurie	Francois Yergeau
Paul J. Leach	Mary Ellen Zurko
Daniel DuBois	

Much of the content and presentation of the caching design is due to suggestions and comments from individuals including: Shel Kaphan, Paul Leach, Koen Holtman, David Morris, and Larry Masinter.

Most of the specification of ranges is based on work originally done by Ari Luotonen and John Franks, with additional input from Steve Zilles.

Thanks to the "cave men" of Palo Alto. You know who you are.

Jim Gettys (the current editor of this document) wishes particularly to thank Roy Fielding, the previous editor of this document, along with John Klensin, Jeff Mogul, Paul Leach, Dave Kristol, Koen Holtman, John Franks, Alex Hopmann, and Larry Masinter for their help.

17 References

- [1] Alvestrand, H., "Tags for the identification of languages", RFC 1766, UNINETT, March 1995.
- [2] Anklesaria, F., McCahill, M., Lindner, P., Johnson, D., Torrey, D., and B. Alberti. "The Internet Gopher Protocol: (a distributed document search and retrieval protocol)", RFC 1436, University of Minnesota, March 1993.
- [3] Berners-Lee, T., "Universal Resource Identifiers in WWW", A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web", RFC 1630, CERN, June 1994.
- [4] Berners-Lee, T., Masinter, L., and M. McCahill, "Uniform Resource Locators (URL)", RFC 1738, CERN, Xerox PARC, University of Minnesota, December 1994.
- [5] Berners-Lee, T., and D. Connolly, "HyperText Markup Language Specification - 2.0", RFC 1866, MIT/LCS, November 1995.
- [6] Berners-Lee, T., Fielding, R., and H. Frystyk, "Hypertext Transfer Protocol -- HTTP/1.0.", RFC 1945 MIT/LCS, UC Irvine, May 1996.
- [7] Freed, N., and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, Innosoft, First Virtual, November 1996.
- [8] Braden, R., "Requirements for Internet hosts - application and support", STD 3, RFC 1123, IETF, October 1989.
- [9] Crocker, D., "Standard for the Format of ARPA Internet Text Messages", STD 11, RFC 822, UDEL, August 1982.

- [10] Davis, F., Kahle, B., Morris, H., Salem, J., Shen, T., Wang, R., Sui, J., and M. Grinbaum. "WAIS Interface Protocol Prototype Functional Specification", (v1.5), Thinking Machines Corporation, April 1990.
- [11] Fielding, R., "Relative Uniform Resource Locators", RFC 1808, UC Irvine, June 1995.
- [12] Horton, M., and R. Adams. "Standard for interchange of USENET messages", RFC 1036, AT&T Bell Laboratories, Center for Seismic Studies, December 1987.
- [13] Kantor, B., and P. Lapsley. "Network News Transfer Protocol." A Proposed Standard for the Stream-Based Transmission of News", RFC 977, UC San Diego, UC Berkeley, February 1986.
- [14] Moore, K., "MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text", RFC 2047, University of Tennessee, November 1996.
- [15] Nebel, E., and L. Masinter. "Form-based File Upload in HTML", RFC 1867, Xerox Corporation, November 1995.
- [16] Postel, J., "Simple Mail Transfer Protocol", STD 10, RFC 821, USC/ISI, August 1982.
- [17] Postel, J., "Media Type Registration Procedure", RFC 2048, USC/ISI, November 1996.
- [18] Postel, J., and J. Reynolds, "File Transfer Protocol (FTP)", STD 9, RFC 959, USC/ISI, October 1985.
- [19] Reynolds, J., and J. Postel, "Assigned Numbers", STD 2, RFC 1700, USC/ISI, October 1994.
- [20] Sollins, K., and L. Masinter, "Functional Requirements for Uniform Resource Names", RFC 1737, MIT/LCS, Xerox Corporation, December 1994.
- [21] US-ASCII. Coded Character Set - 7-Bit American Standard Code for Information Interchange. Standard ANSI X3.4-1986, ANSI, 1986.
- [22] ISO-8859. International Standard -- Information Processing -- 8-bit Single-Byte Coded Graphic Character Sets --
 - Part 1: Latin alphabet No. 1, ISO 8859-1:1987.
 - Part 2: Latin alphabet No. 2, ISO 8859-2, 1987.
 - Part 3: Latin alphabet No. 3, ISO 8859-3, 1988.
 - Part 4: Latin alphabet No. 4, ISO 8859-4, 1988.

- Part 5: Latin/Cyrillic alphabet, ISO 8859-5, 1988.
- Part 6: Latin/Arabic alphabet, ISO 8859-6, 1987.
- Part 7: Latin/Greek alphabet, ISO 8859-7, 1987.
- Part 8: Latin/Hebrew alphabet, ISO 8859-8, 1988.
- Part 9: Latin alphabet No. 5, ISO 8859-9, 1990.

[23] Meyers, J., and M. Rose "The Content-MD5 Header Field", RFC 1864, Carnegie Mellon, Dover Beach Consulting, October, 1995.

[24] Carpenter, B., and Y. Rekhter, "Renumbering Needs Work", RFC 1900, IAB, February 1996.

[25] Deutsch, P., "GZIP file format specification version 4.3." RFC 1952, Aladdin Enterprises, May 1996.

[26] Venkata N. Padmanabhan and Jeffrey C. Mogul. Improving HTTP Latency. Computer Networks and ISDN Systems, v. 28, pp. 25-35, Dec. 1995. Slightly revised version of paper in Proc. 2nd International WWW Conf. '94: Mosaic and the Web, Oct. 1994, which is available at <http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/DDay/mogul/HTTPLatency.html>.

[27] Joe Touch, John Heidemann, and Katia Obraczka, "Analysis of HTTP Performance", <URL: <http://www.isi.edu/lam/ib/http-perf/>>, USC/Information Sciences Institute, June 1996

[28] Mills, D., "Network Time Protocol, Version 3, Specification, Implementation and Analysis", RFC 1305, University of Delaware, March 1992.

[29] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3." RFC 1951, Aladdin Enterprises, May 1996.

[30] Spero, S., "Analysis of HTTP Performance Problems" <URL:<http://sunsite.unc.edu/mdma-release/http-prob.html>>.

[31] Deutsch, P., and J-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3", RFC 1950, Aladdin Enterprises, Info-ZIP, May 1996.

[32] Franks, J., Hallam-Baker, P., Hostetler, J., Leach, P., Luotonen, A., Sink, E., and L. Stewart, "An Extension to HTTP : Digest Access Authentication", RFC 2069, January 1997.

18 Authors' Addresses

Roy T. Fielding
Department of Information and Computer Science
University of California
Irvine, CA 92717-3425, USA

Fax: +1 (714) 824-4056
EMail: fielding@ics.uci.edu

Jim Gettys
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, USA

Fax: +1 (617) 258 8682
EMail: jg@w3.org

Jeffrey C. Mogul
Western Research Laboratory
Digital Equipment Corporation
250 University Avenue
Palo Alto, California, 94305, USA

EMail: mogul@wrl.dec.com

Henrik Frystyk Nielsen
W3 Consortium
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, USA

Fax: +1 (617) 258 8682
EMail: frystyk@w3.org

Tim Berners-Lee
Director, W3 Consortium
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, USA

Fax: +1 (617) 258 8682
EMail: timbl@w3.org

19 Appendices

19.1 Internet Media Type message/http

In addition to defining the HTTP/1.1 protocol, this document serves as the specification for the Internet media type "message/http". The following is to be registered with IANA.

Media Type name: message
Media subtype name: http
Required parameters: none
Optional parameters: version, msgtype

version: The HTTP-Version number of the enclosed message (e.g., "1.1"). If not present, the version can be determined from the first line of the body.

msgtype: The message type -- "request" or "response". If not present, the type can be determined from the first line of the body.

Encoding considerations: only "7bit", "8bit", or "binary" are permitted

Security considerations: none

19.2 Internet Media Type multipart/byteranges

When an HTTP message includes the content of multiple ranges (for example, a response to a request for multiple non-overlapping ranges), these are transmitted as a multipart MIME message. The multipart media type for this purpose is called "multipart/byteranges".

The multipart/byteranges media type includes two or more parts, each with its own Content-Type and Content-Range fields. The parts are separated using a MIME boundary parameter.

Media Type name: multipart
Media subtype name: byteranges
Required parameters: boundary
Optional parameters: none

Encoding considerations: only "7bit", "8bit", or "binary" are permitted

Security considerations: none

For example:

```
HTTP/1.1 206 Partial content
Date: Wed, 15 Nov 1995 06:25:24 GMT
Last-modified: Wed, 15 Nov 1995 04:58:08 GMT
Content-type: multipart/byteranges; boundary=THIS_STRING_SEPARATES
```

```
--THIS_STRING_SEPARATES
Content-type: application/pdf
Content-range: bytes 500-999/8000
```

```
...the first range...
--THIS_STRING_SEPARATES
Content-type: application/pdf
Content-range: bytes 7000-7999/8000
```

```
...the second range
--THIS_STRING_SEPARATES--
```

19.3 Tolerant Applications

Although this document specifies the requirements for the generation of HTTP/1.1 messages, not all applications will be correct in their implementation. We therefore recommend that operational applications be tolerant of deviations whenever those deviations can be interpreted unambiguously.

Clients SHOULD be tolerant in parsing the Status-Line and servers tolerant when parsing the Request-Line. In particular, they SHOULD accept any amount of SP or HT characters between fields, even though only a single SP is required.

The line terminator for message-header fields is the sequence CRLF. However, we recommend that applications, when parsing such headers, recognize a single LF as a line terminator and ignore the leading CR.

The character set of an entity-body should be labeled as the lowest common denominator of the character codes used within that body, with the exception that no label is preferred over the labels US-ASCII or ISO-8859-1.

Additional rules for requirements on parsing and encoding of dates and other potential problems with date encodings include:

- o HTTP/1.1 clients and caches should assume that an RFC-850 date which appears to be more than 50 years in the future is in fact in the past (this helps solve the "year 2000" problem).

- o An HTTP/1.1 implementation may internally represent a parsed Expires date as earlier than the proper value, but MUST NOT internally represent a parsed Expires date as later than the proper value.
- o All expiration-related calculations must be done in GMT. The local time zone MUST NOT influence the calculation or comparison of an age or expiration time.
- o If an HTTP header incorrectly carries a date value with a time zone other than GMT, it must be converted into GMT using the most conservative possible conversion.

19.4 Differences Between HTTP Entities and MIME Entities

HTTP/1.1 uses many of the constructs defined for Internet Mail (RFC 822) and the Multipurpose Internet Mail Extensions (MIME) to allow entities to be transmitted in an open variety of representations and with extensible mechanisms. However, MIME [7] discusses mail, and HTTP has a few features that are different from those described in MIME. These differences were carefully chosen to optimize performance over binary connections, to allow greater freedom in the use of new media types, to make date comparisons easier, and to acknowledge the practice of some early HTTP servers and clients.

This appendix describes specific areas where HTTP differs from MIME. Proxies and gateways to strict MIME environments SHOULD be aware of these differences and provide the appropriate conversions where necessary. Proxies and gateways from MIME environments to HTTP also need to be aware of the differences because some conversions may be required.

19.4.1 Conversion to Canonical Form

MIME requires that an Internet mail entity be converted to canonical form prior to being transferred. Section 3.7.1 of this document describes the forms allowed for subtypes of the "text" media type when transmitted over HTTP. MIME requires that content with a type of "text" represent line breaks as CRLF and forbids the use of CR or LF outside of line break sequences. HTTP allows CRLF, bare CR, and bare LF to indicate a line break within text content when a message is transmitted over HTTP.

Where it is possible, a proxy or gateway from HTTP to a strict MIME environment SHOULD translate all line breaks within the text media types described in section 3.7.1 of this document to the MIME canonical form of CRLF. Note, however, that this may be complicated by the presence of a Content-Encoding and by the fact that HTTP

allows the use of some character sets which do not use octets 13 and 10 to represent CR and LF, as is the case for some multi-byte character sets.

19.4.2 Conversion of Date Formats

HTTP/1.1 uses a restricted set of date formats (section 3.3.1) to simplify the process of date comparison. Proxies and gateways from other protocols **SHOULD** ensure that any Date header field present in a message conforms to one of the HTTP/1.1 formats and rewrite the date if necessary.

19.4.3 Introduction of Content-Encoding

MIME does not include any concept equivalent to HTTP/1.1's Content-Encoding header field. Since this acts as a modifier on the media type, proxies and gateways from HTTP to MIME-compliant protocols **MUST** either change the value of the Content-Type header field or decode the entity-body before forwarding the message. (Some experimental applications of Content-Type for Internet mail have used a media-type parameter of ";conversions=<content-coding>" to perform an equivalent function as Content-Encoding. However, this parameter is not part of MIME.)

19.4.4 No Content-Transfer-Encoding

HTTP does not use the Content-Transfer-Encoding (CTE) field of MIME. Proxies and gateways from MIME-compliant protocols to HTTP **MUST** remove any non-identity CTE ("quoted-printable" or "base64") encoding prior to delivering the response message to an HTTP client.

Proxies and gateways from HTTP to MIME-compliant protocols are responsible for ensuring that the message is in the correct format and encoding for safe transport on that protocol, where "safe transport" is defined by the limitations of the protocol being used. Such a proxy or gateway **SHOULD** label the data with an appropriate Content-Transfer-Encoding if doing so will improve the likelihood of safe transport over the destination protocol.

19.4.5 HTTP Header Fields in Multipart Body-Parts

In MIME, most header fields in multipart body-parts are generally ignored unless the field name begins with "Content-". In HTTP/1.1, multipart body-parts may contain any HTTP header fields which are significant to the meaning of that part.

19.4.6 Introduction of Transfer-Encoding

HTTP/1.1 introduces the Transfer-Encoding header field (section 14.40). Proxies/gateways MUST remove any transfer coding prior to forwarding a message via a MIME-compliant protocol.

A process for decoding the "chunked" transfer coding (section 3.6) can be represented in pseudo-code as:

```
length := 0
read chunk-size, chunk-ext (if any) and CRLF
while (chunk-size > 0) {
    read chunk-data and CRLF
    append chunk-data to entity-body
    length := length + chunk-size
    read chunk-size and CRLF
}
read entity-header
while (entity-header not empty) {
    append entity-header to existing header fields
    read entity-header
}
Content-Length := length
Remove "chunked" from Transfer-Encoding
```

19.4.7 MIME-Version

HTTP is not a MIME-compliant protocol (see appendix 19.4). However, HTTP/1.1 messages may include a single MIME-Version general-header field to indicate what version of the MIME protocol was used to construct the message. Use of the MIME-Version header field indicates that the message is in full compliance with the MIME protocol. Proxies/gateways are responsible for ensuring full compliance (where possible) when exporting HTTP messages to strict MIME environments.

MIME-Version = "MIME-Version" ":" 1*DIGIT "." 1*DIGIT

MIME version "1.0" is the default for use in HTTP/1.1. However, HTTP/1.1 message parsing and semantics are defined by this document and not the MIME specification.

19.5 Changes from HTTP/1.0

This section summarizes major differences between versions HTTP/1.0 and HTTP/1.1.

19.5.1 Changes to Simplify Multi-homed Web Servers and Conserve IP Addresses

The requirements that clients and servers support the Host request-header, report an error if the Host request-header (section 14.23) is missing from an HTTP/1.1 request, and accept absolute URIs (section 5.1.2) are among the most important changes defined by this specification.

Older HTTP/1.0 clients assumed a one-to-one relationship of IP addresses and servers; there was no other established mechanism for distinguishing the intended server of a request than the IP address to which that request was directed. The changes outlined above will allow the Internet, once older HTTP clients are no longer common, to support multiple Web sites from a single IP address, greatly simplifying large operational Web servers, where allocation of many IP addresses to a single host has created serious problems. The Internet will also be able to recover the IP addresses that have been allocated for the sole purpose of allowing special-purpose domain names to be used in root-level HTTP URLs. Given the rate of growth of the Web, and the number of servers already deployed, it is extremely important that all implementations of HTTP (including updates to existing HTTP/1.0 applications) correctly implement these requirements:

- o Both clients and servers **MUST** support the Host request-header.
- o Host request-headers are required in HTTP/1.1 requests.
- o Servers **MUST** report a 400 (Bad Request) error if an HTTP/1.1 request does not include a Host request-header.
- o Servers **MUST** accept absolute URIs.

19.6 Additional Features

This appendix documents protocol elements used by some existing HTTP implementations, but not consistently and correctly across most HTTP/1.1 applications. Implementers should be aware of these features, but cannot rely upon their presence in, or interoperability with, other HTTP/1.1 applications. Some of these describe proposed experimental features, and some describe features that experimental deployment found lacking that are now addressed in the base HTTP/1.1 specification.

19.6.1 Additional Request Methods

19.6.1.1 PATCH

The PATCH method is similar to PUT except that the entity contains a list of differences between the original version of the resource identified by the Request-URI and the desired content of the resource after the PATCH action has been applied. The list of differences is in a format defined by the media type of the entity (e.g., "application/diff") and MUST include sufficient information to allow the server to recreate the changes necessary to convert the original version of the resource to the desired version.

If the request passes through a cache and the Request-URI identifies a currently cached entity, that entity MUST be removed from the cache. Responses to this method are not cachable.

The actual method for determining how the patched resource is placed, and what happens to its predecessor, is defined entirely by the origin server. If the original version of the resource being patched included a Content-Version header field, the request entity MUST include a Derived-From header field corresponding to the value of the original Content-Version header field. Applications are encouraged to use these fields for constructing versioning relationships and resolving version conflicts.

PATCH requests must obey the message transmission requirements set out in section 8.2.

Caches that implement PATCH should invalidate cached responses as defined in section 13.10 for PUT.

19.6.1.2 LINK

The LINK method establishes one or more Link relationships between the existing resource identified by the Request-URI and other existing resources. The difference between LINK and other methods

allowing links to be established between resources is that the LINK method does not allow any message-body to be sent in the request and does not directly result in the creation of new resources.

If the request passes through a cache and the Request-URI identifies a currently cached entity, that entity **MUST** be removed from the cache. Responses to this method are not cachable.

Caches that implement LINK should invalidate cached responses as defined in section 13.10 for PUT.

19.6.1.3 UNLINK

The UNLINK method removes one or more Link relationships from the existing resource identified by the Request-URI. These relationships may have been established using the LINK method or by any other method supporting the Link header. The removal of a link to a resource does not imply that the resource ceases to exist or becomes inaccessible for future references.

If the request passes through a cache and the Request-URI identifies a currently cached entity, that entity **MUST** be removed from the cache. Responses to this method are not cachable.

Caches that implement UNLINK should invalidate cached responses as defined in section 13.10 for PUT.

19.6.2 Additional Header Field Definitions

19.6.2.1 Alternates

The Alternates response-header field has been proposed as a means for the origin server to inform the client about other available representations of the requested resource, along with their distinguishing attributes, and thus providing a more reliable means for a user agent to perform subsequent selection of another representation which better fits the desires of its user (described as agent-driven negotiation in section 12).

The Alternates header field is orthogonal to the Vary header field in that both may coexist in a message without affecting the interpretation of the response or the available representations. It is expected that Alternates will provide a significant improvement over the server-driven negotiation provided by the Vary field for those resources that vary over common dimensions like type and language.

The Alternates header field will be defined in a future specification.

19.6.2.2 Content-Version

The Content-Version entity-header field defines the version tag associated with a rendition of an evolving entity. Together with the Derived-From field described in section 19.6.2.3, it allows a group of people to work simultaneously on the creation of a work as an iterative process. The field should be used to allow evolution of a particular work along a single path rather than derived works or renditions in different representations.

Content-Version = "Content-Version" ":" quoted-string

Examples of the Content-Version field include:

Content-Version: "2.1.2"
Content-Version: "Fred 19950116-12:26:48"
Content-Version: "2.5a4-omega7"

19.6.2.3 Derived-From

The Derived-From entity-header field can be used to indicate the version tag of the resource from which the enclosed entity was derived before modifications were made by the sender. This field is used to help manage the process of merging successive changes to a resource, particularly when such changes are being made in parallel and from multiple sources.

Derived-From = "Derived-From" ":" quoted-string

An example use of the field is:

Derived-From: "2.1.1"

The Derived-From field is required for PUT and PATCH requests if the entity being sent was previously retrieved from the same URI and a Content-Version header was included with the entity when it was last retrieved.

19.6.2.4 Link

The Link entity-header field provides a means for describing a relationship between two resources, generally between the requested resource and some other resource. An entity MAY include multiple Link values. Links at the metainformation level typically indicate relationships like hierarchical structure and navigation paths. The Link field is semantically equivalent to the <LINK> element in HTML. [5]

Link = "Link" ":" #("<" URI ">" *(";" link-param)

link-param = (("rel" "=" relationship)
 | ("rev" "=" relationship)
 | ("title" "=" quoted-string)
 | ("anchor" "=" "<" URI ">")
 | (link-extension))

link-extension = token ["=" (token | quoted-string)]

relationship = sgml-name
 | ("<" sgml-name *(SP sgml-name) ">")

sgml-name = ALPHA *(ALPHA | DIGIT | "." | "-")

Relationship values are case-insensitive and MAY be extended within the constraints of the sgml-name syntax. The title parameter MAY be used to label the destination of a link such that it can be used as identification within a human-readable menu. The anchor parameter MAY be used to indicate a source anchor other than the entire current resource, such as a fragment of this resource or a third resource.

Examples of usage include:

Link: <http://www.cern.ch/TheBook/chapter2>; rel="Previous"

Link: <mailto:timbl@w3.org>; rev="Made"; title="Tim Berners-Lee"

The first example indicates that chapter2 is previous to this resource in a logical navigation path. The second indicates that the person responsible for making the resource available is identified by the given e-mail address.

19.6.2.5 URI

The URI header field has, in past versions of this specification, been used as a combination of the existing Location, Content-Location, and Vary header fields as well as the future Alternates

field (above). Its primary purpose has been to include a list of additional URIs for the resource, including names and mirror locations. However, it has become clear that the combination of many different functions within this single field has been a barrier to consistently and correctly implementing any of those functions. Furthermore, we believe that the identification of names and mirror locations would be better performed via the Link header field. The URI header field is therefore deprecated in favor of those other fields.

URI-header = "URI" ":" 1#("<" URI ">")

19.7 Compatibility with Previous Versions

It is beyond the scope of a protocol specification to mandate compliance with previous versions. HTTP/1.1 was deliberately designed, however, to make supporting previous versions easy. It is worth noting that at the time of composing this specification, we would expect commercial HTTP/1.1 servers to:

- o recognize the format of the Request-Line for HTTP/0.9, 1.0, and 1.1 requests;
- o understand any valid request in the format of HTTP/0.9, 1.0, or 1.1;
- o respond appropriately with a message in the same major version used by the client.

And we would expect HTTP/1.1 clients to:

- o recognize the format of the Status-Line for HTTP/1.0 and 1.1 responses;
- o understand any valid response in the format of HTTP/0.9, 1.0, or 1.1.

For most implementations of HTTP/1.0, each connection is established by the client prior to the request and closed by the server after sending the response. A few implementations implement the Keep-Alive version of persistent connections described in section 19.7.1.1.

19.7.1 Compatibility with HTTP/1.0 Persistent Connections

Some clients and servers may wish to be compatible with some previous implementations of persistent connections in HTTP/1.0 clients and servers. Persistent connections in HTTP/1.0 must be explicitly negotiated as they are not the default behavior. HTTP/1.0 experimental implementations of persistent connections are faulty, and the new facilities in HTTP/1.1 are designed to rectify these problems. The problem was that some existing 1.0 clients may be sending Keep-Alive to a proxy server that doesn't understand Connection, which would then erroneously forward it to the next inbound server, which would establish the Keep-Alive connection and result in a hung HTTP/1.0 proxy waiting for the close on the response. The result is that HTTP/1.0 clients must be prevented from using Keep-Alive when talking to proxies.

However, talking to proxies is the most important use of persistent connections, so that prohibition is clearly unacceptable. Therefore, we need some other mechanism for indicating a persistent connection is desired, which is safe to use even when talking to an old proxy that ignores Connection. Persistent connections are the default for HTTP/1.1 messages; we introduce a new keyword (Connection: close) for declaring non-persistence.

The following describes the original HTTP/1.0 form of persistent connections.

When it connects to an origin server, an HTTP client MAY send the Keep-Alive connection-token in addition to the Persist connection-token:

Connection: Keep-Alive

An HTTP/1.0 server would then respond with the Keep-Alive connection token and the client may proceed with an HTTP/1.0 (or Keep-Alive) persistent connection.

An HTTP/1.1 server may also establish persistent connections with HTTP/1.0 clients upon receipt of a Keep-Alive connection token. However, a persistent connection with an HTTP/1.0 client cannot make use of the chunked transfer-coding, and therefore MUST use a Content-Length for marking the ending boundary of each message.

A client MUST NOT send the Keep-Alive connection token to a proxy server as HTTP/1.0 proxy servers do not obey the rules of HTTP/1.1 for parsing the Connection header field.

19.7.1.1 The Keep-Alive Header

When the Keep-Alive connection-token has been transmitted with a request or a response, a Keep-Alive header field MAY also be included. The Keep-Alive header field takes the following form:

Keep-Alive-header = "Keep-Alive" ":" 0# keepalive-param

keepalive-param = param-name "=" value

The Keep-Alive header itself is optional, and is used only if a parameter is being sent. HTTP/1.1 does not define any parameters.

If the Keep-Alive header is sent, the corresponding connection token MUST be transmitted. The Keep-Alive header MUST be ignored if received without the connection token.



P.B.5818 - Patentlaan 2
2280 HV Rijswijk (ZH)
☎ +31 70 340 2040
TX 31651 epo nl
FAX +31 70 340 3016

FILED BY IDS
04-30-02

Europäisches
Patentamt

Zweigstelle
in Den Haag
Recherchen-
abteilung

European
Patent Office

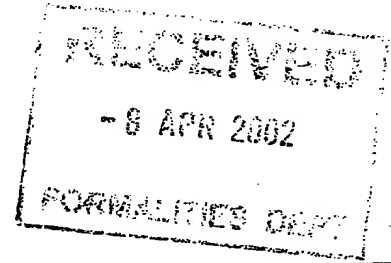
Branch at
The Hague
Search
division

Office européen
des brevets

Département à
La Haye
Division de la
recherche

Luckhurst, Anthony Henry William
MARKS & CLERK,
57-60 Lincoln's Inn Fields
London WC2A 3LS
GRANDE BRETAGNE

INPROMA UPDATED
Date: 8/4/02
Initialed: SH



Datum/Date
09.04.02

Zeichen/Ref./Réf.

EPP82049

Anmeldung Nr./Application No./Demande n°/Patent Nr./Patent No./Brevet n°.
00302367.8-2201-

Anmelder/Applicant/Demandeur/Patentinhaber/Proprietor/Titulaire
KABUSHIKI KAISHA TOSHIBA

COMMUNICATION

The European Patent Office herewith transmits as an enclosure the European search report for the above-mentioned European patent application.

If applicable, copies of the documents cited in the European search report are attached.

☒ Additional set(s) of copies of the documents cited in the European search report is (are) enclosed as well.

The following specifications given by the applicant have been approved by the Search Division:

☒ abstract ☒ title

☐ The abstract was modified by the Search Division and the definitive text is attached to this communication.

The following figure will be published together with the abstract: 5

REFUND OF THE SEARCH FEE

If applicable under Article 10 Rules relating to fees, a separate communication from the Receiving Section on the refund of the search fee will be sent later.





DOCUMENTS CONSIDERED TO BE RELEVANT															
Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.7)												
X	AU 53031 98 A (MILLS DUDLEY JOHN) 27 August 1998 (1998-08-27) * page 6, line 40 - page 9, line 45 * * page 15, line 26 - page 22, line 3 * ---	1-20	G06F17/30												
X	US 5 715 397 A (OGAWA STUART S ET AL) 3 February 1998 (1998-02-03) * the whole document *	1-20													
X	US 5 629 846 A (CRAPO ANDREW W) 13 May 1997 (1997-05-13) * the whole document *	1-20													
X	US 5 627 997 A (BRAY BRIAN D ET AL) 6 May 1997 (1997-05-06) * column 2, line 23 - column 8, line 48 *	1-20													
X	US 5 557 780 A (SHIRROD REBECCA K ET AL) 17 September 1996 (1996-09-17) * the whole document *	1-20													
X	NADO, R. A., HUFFMAN, S.B.: "Extracting Entity Profiles from Semistructured Information Spaces" ACM SIGMOD RECORD, vol. 26, no. 4, December 1997 (1997-12), pages 32-38, XP002193377 USA ISSN 0163-5808 * the whole document * --- -/--	1-20													
The present search report has been drawn up for all claims															
Place of search MUNICH		Date of completion of the search 18 March 2002	Examiner Jaedicke, M												
<table border="0"><tr><td>CATEGORY OF CITED DOCUMENTS</td><td>T : theory or principle underlying the invention</td></tr><tr><td>X : particularly relevant if taken alone</td><td>E : earlier patent document, but published on, or after the filing date</td></tr><tr><td>Y : particularly relevant if combined with another document of the same category</td><td>D : document cited in the application</td></tr><tr><td>A : technological background</td><td>L : document cited for other reasons</td></tr><tr><td>O : non-written disclosure</td><td>& : member of the same patent family, corresponding document</td></tr><tr><td>P : intermediate document</td><td></td></tr></table>				CATEGORY OF CITED DOCUMENTS	T : theory or principle underlying the invention	X : particularly relevant if taken alone	E : earlier patent document, but published on, or after the filing date	Y : particularly relevant if combined with another document of the same category	D : document cited in the application	A : technological background	L : document cited for other reasons	O : non-written disclosure	& : member of the same patent family, corresponding document	P : intermediate document	
CATEGORY OF CITED DOCUMENTS	T : theory or principle underlying the invention														
X : particularly relevant if taken alone	E : earlier patent document, but published on, or after the filing date														
Y : particularly relevant if combined with another document of the same category	D : document cited in the application														
A : technological background	L : document cited for other reasons														
O : non-written disclosure	& : member of the same patent family, corresponding document														
P : intermediate document															

1
EPO FORM 1503 03.82 (P04C01)



DOCUMENTS CONSIDERED TO BE RELEVANT			CLASSIFICATION OF THE APPLICATION (Int.Cl.7)
Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	
X	ADELBERG B: "NODOSE - A TOOL FOR SEMI-AUTOMATICALLY EXTRACTING STRUCTURED AND SEMISTRUCTURED DATA FROM TEXT DOCUMENTS" ACM PROCEEDINGS OF SIGMOD. INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, XX, XX, vol. 27, no. 2, 1998, pages 1-25, XP002949327 * page 20, paragraph 4.2 - page 22, line 2; figure 1 *	1-20	
X	ASHISH N ET AL: "Semi-automatic wrapper generation for Internet information sources" COOPERATIVE INFORMATION SYSTEMS, 1997. COOPIS '97., PROCEEDINGS OF THE SECOND IFCIS INTERNATIONAL CONFERENCE ON KIAWAH ISLAND, SC, USA 24-27 JUNE 1997, LOS ALAMITOS, CA, USA, IEEE COMPUT. SOC, US, 24 June 1997 (1997-06-24), pages 160-169, XP010240791 ISBN: 0-8186-7946-8 * the whole document *	1-20	
A	FLORESCU, D., LEVY, A., MENDELZON, A.: "Database Techniques for the World-Wide Web: A Survey" ACM SIGMOD RECORD, vol. 27, no. 3, September 1998 (1998-09), pages 59-74, XP002193378 USA * the whole document *	1-20	
A	WO 98 03928 A (LEXTRON SYSTEMS INC) 29 January 1998 (1998-01-29) * the whole document *	1-20	
The present search report has been drawn up for all claims			
Place of search MUNICH		Date of completion of the search 18 March 2002	Examiner Jaedicke, M
<p>CATEGORY OF CITED DOCUMENTS</p> <p>X : particularly relevant if taken alone Y : particularly relevant if combined with another document of the same category A : technological background O : non-written disclosure P : intermediate document</p> <p>T : theory or principle underlying the invention E : earlier patent document, but published on, or after the filing date D : document cited in the application L : document cited for other reasons & : member of the same patent family, corresponding document</p>			

1
EPO FORM 1503 03.82 (P04C01)



DOCUMENTS CONSIDERED TO BE RELEVANT			
Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.7)
A	US 5 835 712 A (DUFRESNE FRED B) 10 November 1998 (1998-11-10) * the whole document *	1-20	
A	US 5 721 912 A (MATERNA ANTHONY T ET AL) 24 February 1998 (1998-02-24) * column 9, line 10 - column 10, line 32 *	1-20	
			TECHNICAL FIELDS SEARCHED (Int.Cl.7)
The present search report has been drawn up for all claims			
Place of search MUNICH		Date of completion of the search 18 March 2002	Examiner Jaedicke, M
<div>CATEGORY OF CITED DOCUMENTS</div> <div><div>X : particularly relevant if taken alone Y : particularly relevant if combined with another document of the same category A : technological background O : non-written disclosure P : intermediate document</div><div>T : theory or principle underlying the invention E : earlier patent document, but published on, or after the filing date D : document cited in the application L : document cited for other reasons & : member of the same patent family, corresponding document</div></div>			

ANNEX TO THE EUROPEAN SEARCH REPORT ON EUROPEAN PATENT APPLICATION NO.

EP 00 30 2367

This annex lists the patent family members relating to the patent documents cited in the above-mentioned European search report. The members are as contained in the European Patent Office EDP file on
The European Patent Office is in no way liable for these particulars which are merely given for the purpose of information.

18-03-2002

Patent document cited in search report		Publication date	Patent family member(s)	Publication date
AU 5303198	A	27-08-1998	AU 740007 B2 US 2001021935 A1	25-10-2001 13-09-2001
US 5715397	A	03-02-1998	US 5608874 A	04-03-1997
US 5629846	A	13-05-1997	NONE	
US 5627997	A	06-05-1997	NONE	
US 5557780	A	17-09-1996	NONE	
WO 9803928	A	29-01-1998	WO 9803928 A1	29-01-1998
US 5835712	A	10-11-1998	AU 2826797 A EP 0978055 A1 WO 9742584 A1	26-11-1997 09-02-2000 13-11-1997
US 5721912	A	24-02-1998	NONE	

ORIGINAL

AUSTRALIA

Patents Act 1990

COMPLETE SPECIFICATION

APPLICANT: Dudley John MILLS

ADDRESS: 30 Hutchison Crescent,
Kambah, ACT 2902

ACTUAL INVENTOR: Dudley John MILLS

ADDRESS FOR SERVICE: Dudley J. Mills,
30 Hutchison Crescent,
Kambah, ACT2902

ASSOCIATED PROVISIONAL: PO5254 Filed 21 February 1997

INVENTION TITLE: NETWORK-BASED CLASSIFIED
INFORMATION SYSTEMS

The following is a full description of the invention including the best method of performing it known to me:

TITLE: NETWORK BASED CLASSIFIED INFORMATION SYSTEMS

FIELD OF INVENTION

This invention relates to network based classified information systems, to methods of automatically building searchable databases of classified information derived from web pages posted on a network, and, to web pages for use in such systems and methods.

The information systems and databases of most relevance to this invention are those which include classified product and service catalogues similar to the Yellow Pages telephone books, contact indexes similar to the White Pages telephone books, and/or subject indexes similar to Library catalogues. Such information systems and databases typically include sets of associated classification, contact and/or geographic items of information. For convenience, classification, contact and/or geographic information will be hereinafter called CCG-data.

The networks with which this invention is concerned are the worldwide public computer/communications network commonly known as the Internet and private networks – sometimes called intranets – which allow common access to markup documents on computers connected to the network. Markup documents are text files prepared using various markup languages such as HyperText Markup Language (HTML) and Extensible Markup Language (XML) which are implementations (or dialects) of the Standard Generalised Markup Language (SGML). The system of accessible files on the Internet is called the World Wide Web (WWW) and the markup documents themselves are commonly called 'web pages'. A web page is said to be 'posted' on a network when it is stored on computer-readable media of a host network computer as a file which is generally accessible to network users. A web page is transported from the host computer to a requesting computer through intermediate network computers as a computer-readable signal embodied in a carrier wave. Though this invention is not limited to Internet based information systems, these terms are used for convenience.

BACKGROUND TO THE INVENTION

It has been estimated that there are about 100 million web pages on the Internet and that the number is doubling every two years. Many of these pages include information concerning commercially offered goods and services and often include contact details. But the difficulty of locating such information is increasing faster than the growth in the number of web pages.

To assist network users locate web pages of interest, certain network service providers create indexes (or databases) of the contents of web pages posted (stored on computer readable media so as to be generally accessible) on the network and provide 'search engines' to use the indexes. These indexes are often created automatically by the use of 'web crawlers' which (i) interrogate computer after computer on the network to locate successive web pages and (ii) index the words in each web page encountered against the network address (eg Internet Protocol Address or IPA) and filing system path or universal resource locator (URL) at which the web page is accessible. Hereinafter the terms URL and URI (Uniform Resource Identifier) are taken to be identical in meaning and to signify network addresses and filing system paths. Usually, the indexes consist of a list of unique words with each word having an associated list of URLs of the web pages wherein the word was found to occur during interrogation. The URL serves as a 'hyperlink' which, if selected by a user/searcher, results in the associated web page being automatically transmitted from the computer where it is posted on the network to the user/searcher's computer where it may be displayed or otherwise processed. The sending and receiving of files in this way is greatly assisted by user interface programs called 'web browsers' (or more simply, 'browsers') such as Netscape and Microsoft Internet Explorer.

The search for web pages of interest using search engines leaves much to be desired:

- simple searches (those using a few keywords in simple combinations) often yield far too many web page references (URLs) to permit them to be interrogated one-by-one.
- 5 • complex searches (those using many keywords and/or complex Boolean expressions) require considerable expertise to undertake,
- even using optimum search criteria, many irrelevant web pages are referenced because of inconsistent use of terminology by those who author the original web pages,
- even using optimum search criteria, many relevant pages are missed, again because of
- 10 • inconsistent use of terminology by web page authors, and
- because items of information included in the body of web pages cannot be 'understood' or associated in useful ways by web crawlers; that is recognised as, say, a surname, a street name, a geographic locality, or type of goods or services and, say, a surname strongly associated with a street name, a geographic locality, or a type of goods or service.
- 15 The result is that information provided by search engines from databases which are automatically compiled using web crawlers is a very poor equivalent of the common Yellow Pages and White Pages directories which serve the telephone industry (though these directories are not, of course, automatically compiled from web pages).
- 20 In an attempt to improve the usefulness of automatically compiled network databases, some search engine providers make use of information contained in URLs, such as the country code and top level domain name codes such as 'com', 'edu', 'net' and 'org' which is sometimes used to signify the subject matter of web pages. It has been proposed to add more content classifying codes to URLs (eg, "chem" to signify chemical subject matter) to allow specialised
- 25 databases – national, commercial, chemical, etc – to be generated. However, this proposal has serious drawbacks:
 - URLs are Internet addresses and it is in principle undesirable to confuse the address function of a URL with that of representing a list of web page classifications or contact details.
 - 30 • A URL is an inappropriate container of multiple web page classification codes and contact details because the length of the URL would cause it to become unwieldy as an Internet address.
 - Including in a URL classification codes drawn from a list of thousands of codes would compromise the mnemonic quality of Internet addresses such as "www.yellowpages.com".
 - 35 • There is substantial overlap in the subject matter contained in web pages having the various top level domain name codes.
 - There is no consensus on, or standard for, content classification codes in URLs.

- Another proposal to add content classification data to web pages has arisen from the wish to
- 40 identify pages containing material that may be offensive to some viewers, or should not be accessed by minors. The Platform for Internet Content Selection (PICS) (see <http://www.w3.org/pub/WWW/PICS> and other documents at www.w3.org) is a web page ratings standard similar in principle to the ratings systems for motion pictures. This system allows page authors to "internally" self classify their pages through use of the "<meta...>"
- 45 HTML element. Alternatively, "external" PICS ratings of web pages may be obtained from ratings service providers accessed each time a URL is selected. In practice, the ratings service providers have adopted very limited range of web page classifications. For example, Ararat Software's Commercial Rating System (see <http://www.ararat.com.ratings/ararat10.html>) provides just 5 categories of web page content: commercial content, technical/customer
- 50 support, ordering information, downloading information and contact information. In other

examples, CyberPatrol (http://www.microsys.com/pics/pics_msi.htm) provides 16 categories, the Recreational Software Advisory Council (<http://www.rsac.org/faq.html>) provides 4 categories, SafeSurf (<http://www.safesurf.com/ssplan.htm>) provides 11 categories and Vancouver Webpages Rating Service (<http://vancouver-webpages.com/WWP1.0/>) provides 11 categories. None of the categories provide classification of web pages by industry, service, product or subject with sufficient specificity to be useful when searching for web pages. Rather, the categories are intended to prevent web browsers from displaying web pages unsuitable for particular types of web browser users. Such rating systems are not intended to be used for the automated creation of Yellow or White pages like databases from web pages and are unsuitable for that purpose because they can not represent contact details. Further, the ratings data may only be encoded in the <meta...> element in the <head> of an HTML document drastically limiting the type and usefulness of the data that can be encoded.

Another proposal for classifying the content of web pages, the "Meta Content Framework" (MCF - see <http://mcf.research.apple.com/mcf.html>), requires the content of web pages to be classified and the classification data to be held in a separate non-HTML data file with a MIME type of text/mcf. Storing data in non-HTML encoded documents which describes the content of HTML encoded documents is a technical and economic barrier to the adoption by search engine providers of the proposal. The MCF proposal is thus entirely unsuited to the automated creation of Yellow or White pages like databases from HTML encoded web pages (MIME type text/html) because data stored according to the MCF proposal is not stored in HTML encoded web pages.

The "Electronic Business Card", vCard, (see "vCard The Electronic Business Card" Version 2.1, versit Consortium Specification, Sept 18, 1996 or <ftp://ds.internic.net/internet-drafts/draft-ietf-asid-mime-vcard-01.txt>) uses non-HTML data file (MIME Content Types of "text/plain" or the non-standard "text/X-vCard") containing contact information equivalent to an extended White Pages entry which can be exchanged on a network using Simple Mail Transfer Protocol (SMTP) or using HTTP. It can be associated with a web page by use of a URL in the web page which refers to the vCard information (eg My vCard). Version 2.1 vCard standard data file format (published 18 September 1996) provides for the inclusion of many items of contact information. The vCard specification recommends that, where possible, there should be consistent mapping of vCard property names to HTML "<input>" element attribute names (eg vCard property name "TITLE" maps to HTML "<input name= 'title'>"). The intention is to facilitate the transfer of vCard data into web page input forms by pasting from a clipboard or by dragging from other computer applications. The VCard proposal is unsuited to the automated creation of Yellow or White pages like databases from HTML encoded web pages because data stored according to the VCard proposal is not stored in HTML encoded web pages.

The inclusion of classified information in separate documents (such as Meta Content files or vCards) has the disadvantage that there is necessarily much duplication of data and coordination of modifications between the separate documents and the web pages. This must be done to allow a person who has accessed a web page using an HTML compliant browser to determine whether it is worth calling up the associated file or vice versa. Also, to allow portions of web pages to be classified, web page contextual information would have to be duplicated in the separate document. vCards in particular do not provide this functionality. Another disadvantage is that non-HTML documents such as vCards contain no details as to how the data they contain is to be displayed. In the display of HTML documents the position, font, size, colour of the text and other elements of the document are of great importance. The

restriction of address data in a vCard to untagged ordinally organised fields is inflexible. For example, multiple instances of extended parts of the address are not possible. Also components of names, addresses and telephone numbers and so forth are insufficiently identified.

5

- The Online Computer Library Center Inc (OCLC, Dublin, Ohio, USA) proposal, known as the "Dublin Core", proposes to classifying scholarly web pages by subject (topic of the work, or keywords that describe the content of the work), title, author, publisher, other agent, date, object type (genre of the object such as home page, novel, poem etc), form, identifier, source, language, relationship and coverage (spatial and temporal) (see <http://www.oclc.org:5046/~weibel/html-meta.html> and other documents at www.oclc.org). This proposal does not include industry, service, product or subject classifications. It also does not include contact details. Names such as that of the author are not specified in sufficient detail to avoid ambiguities such as which is the author's first and last names. The proposal specifies that the details are encoded using the <meta...> element in the <head> of web pages. The proposal is unsuited to the automated creation of Yellow or White pages like databases from web pages because the proposal does not provide for classification of web pages and does not provide adequate contact details. Further, the use of keywords for describing the content of the work adds very little to the effectiveness of indexing of web pages since the web pages are usually indexed on every word of their content and most often the key words would simply be a duplication of words already contained in the document.

- It has also been proposed to use the Dewey Decimal System (see http://orc.rsch.oclc.org:6109/eval_dc.html and <http://orc.rsch.oclc.org:6109/bintrio.html>) to rank electronic documents against a Dewey Decimal subject classification. The proposal suggests automatically assigning Dewey Decimal subject classification codes to documents during automated indexing and cataloguing but does not specify the exact nature of the assignment although it is implied that the codes are stored separately from the documents. The proposal admits that such automated classification is less satisfactory than human classification. The proposal is unsuited to the automated creation of Yellow or White pages like databases from web pages because the accuracy of classification is inadequate, does not provide for inclusion of industry, service or product classifications and does not provide for inclusion of contact details. Deriving a subject classification code from an analysis of every word and phrase in a web page is computationally expensive.

35

- The HTML 3.0 standard (see page 23 of the www.w3.org document "draft-ietf-html-specv3-00.txt") provides "class" as an attribute of almost all HTML "<body>" elements. The "class" attribute is intended to be used with style sheets. Style sheets provide a means by which the display of HTML documents may be altered to suit the needs of different classes of browser users. For example, <div class="appendix"> could be used to define a division that acts as an appendix, <h2 class="section"> could be used to define a level 2 header that acts as a section header, although, of course, any string of characters could be defined for those purposes. The "class" attribute, although never having been suggested for holding goods and services classifications, is not suited for such a use as it is, in any case, undesirable to confuse the style sheet function of the "class" attribute.

- The HTML 3.0 and earlier standards provided the HTML elements "<person>" and "<address>" but do not specify the form of the content or method of validating the content of those elements. A person's name may be written as first name followed by last name or last name followed by first name. Similarly, different conventions exist for writing addresses. Similar

ambiguities arise in the ill defined format of the HTML elements "<person>" and "<address>". As such they are of little use in the automatic compilation of searchable databases.

The XML language (see: <http://textuality.com/sgml-erb/WD-xml.html>) was developed to extend HTML so that software vendors can add new elements and new element attributes to HTML which are not specifically defined in any HTML standard. The intention is to ensure that all new elements and attributes could be parsed by all XML parsers even if the new elements held no significance for any particular XML parser. However, like HTML, XML does not provide a standard for the representation of industry, service, product or subject classification, contact or geographic location details within an web page.

Of course, many useful databases of the Yellow Pages or White Pages type are made available by service providers on networks, but they are not compiled automatically by using web crawlers to scan HTML web pages posted on a network. For example, <http://www.yellowpages.com.au> and <http://www.mcp.com> provide classified advertisements of the Yellow Pages type with links to the web pages of paying advertisers or subscribers. There are also directories of email addresses which approximate the White Pages directories, listing the names of individuals and organisations and contact details, (eg <http://www.bigbook.com> and <http://query1.whowhere.com>). However, these email directories require listers to manually add their directory entries and enquirers to be aware of and to find the directory enquiry web page. They cannot be automatically generated by scanning web pages using web crawlers since there is no adequate mechanism to relate email addresses to the names of people and organisations and their other contact details which may also exist in the same web page.

25 OBJECTIVES OF THE INVENTION

The general object of the invention is to provide improved methods for automatically building searchable databases of classification, contact, and/or geographical information by using web crawlers to interrogate web pages posted on a network. [For convenience, this information is collectively referred to as CCG-data].

Other non-essential objectives are to provide methods for including and/or displaying CCG-data within web pages accessed by browsers, for automatically extracting CCG-data from web pages posted on a network and for using the same, and/or to provide methods for searching automatically compiled databases using such data.

Another subsidiary objective of the invention is to provide a new form of web page which is better suited to the automatic compilation (using web crawlers) of databases constructed by the automatic scanning of many such pages posted on a network.

40 OUTLINE OF THE INVENTION

The invention is based upon the realisation that highly useful databases can be automatically built by successively interrogating web pages posted on a network if one or more HTML encoded CCG phrases are included in the web pages. A CCG phrase is one containing CCG-data in a form which is directly accessible and identifiable. CCG phrases may also include one or more items which provide the web page author with control over how the CCG-data is applied to the database.

Data duplication can be reduced if some of the CCG-data in the coded CCG phrases can be displayed by browsers as well as being used to update databases. Errors due to inexact duplicated data are also eliminated. Accordingly, it is envisaged that CCG phrases may include

one or more items which provide the web page author with control over how the CCG-data is displayed by a browser.

HTML (including version 2 and version 3) and XML are evolving applications (sub-sets or
5 dialects) of ISO Standard 8879 1986 known as Standard Generalised Markup Language (SGML). HTML, in large part, is a language used to describe how text (unstructured data) and graphics is to be formatted for display. The HTML language consists of a finite number of "elements" (for example; "
" where "BR" is the element name, also called the tag name) which may contain "attributes" (for example; "<DL COMPACT>" where "COMPACT" is an
10 attribute named "COMPACT") and may contain values associated with attributes (for example; "" where +1 is the attribute value of the attribute named "SIZE"). XML is a language used to describe structured data. The XML language is similarly composed of elements, attributes and values with a similar syntax to HTML but unlike HTML the element names which may be used are not restricted and the meaning of the XML data may be
15 interpreted in any convenient manner. While the XML language is mute about how data described by XML is to be formatted for display, the data may be used by computer programs for any purpose including description of how XML coded data is displayed. However, due to its historic importance in connection with web pages, the term "HTML" is herein used to refer to all markup languages which are subsets or complete sets of the SGML language. In particular,
20 the term "HTML encoded CCG phrase" and the synonymous term "CCG phrase" are herein used to refer to CCG-data encoded in a subset or complete set of the SGML language. Herein, a "web page" is a document adapted to be or actually accessible through a network and encoded in a subset or complete set of the SGML language.

25 For convenience, CCG items in HTML encoded CCG phrases, whether they are syntactically represented as elements or as attributes, will be referred to hereinafter as CCG attributes.

A CCG phrase includes at least one of the following identifiable types of CCG-data attributes:

- industry, product, service, and/or subject classifications,
- 30 • contact categories, contact person(s) and/or organisation(s) names, titles or associations, contact details including physical and postal addresses, telephone and fax numbers, email and Internet or network addresses or locations, public keys, and
- geographic location details.

35 A CCG phrase may also include any of the following identifiable types of CCG control attributes:

- database control attributes to indicate which parts of the data are to be used to update databases, and
- display control attributes to indicate how browsers are to display the data.

40

By virtue of occurring in the same CCG phrase, a plurality of CCG-data attributes are associated with each other.

By virtue of their occurrence in the same CCG phrase, CCG-data attributes are identified as
45 a set of associated attributes. However the degree of association between attributes can be controlled by the inclusion in the phrase of database control attributes.

The start and end of CCG phrases should be identifiable to clearly distinguish these phrases from other data. To identify the beginning and end of a CCG phrase, at least one HTML
50 element should have a CCG specific HTML element name or CCG specific attribute name or

CCG specific value. Each CCG attribute may consist, with or without other incidental characters, of a CCG attribute name and/or a CCG value or values. Preferably, each CCG phrase is contained in the "<body>" of the web page.

- 5 Two examples of a CCG specific HTML element are: "<CCG ...>" or "<CCG ... />" or "<CCG>...</CCG>". (Where a CCG phrase is coded in XML, the elements "<XML>" and "</XML>" may also be needed at the start and end of the CCG phrase.) A less satisfactory example is: "<!--CCG ...-->" where the characters "CCG" after HTML comment element name "!--" are used to signify that the comment contains CCG-data. An example of the use of a CCG
- 10 specific attribute name is: "<START CCG>..."<END CCG>". An example of the use of a CCG specific value is: "<START TYPE='CCG'>..."<END TYPE='CCG'>". Obviously, other character strings could be substituted for the element name, element attribute name or element attribute value "CCG" string of the examples.
- 15 The codes "<CCG ...>" and "<CCG ... />" are compatible with most HTML specifications, but being non-standard HTML, most web browsers do not display any text or attributes (eg PQ="AQD") within the angle brackets "<" and ">". These codes are preferred where display of the CCG data is not required and compatibility with older browsers is required (eg CCG phrases containing only classification values).
- 20 From one aspect, therefore, the invention comprises a web page for posting on a network, the web page being characterised by the inclusion of at least one CCG phrase in the "<body>" of the page, the CCG phrase being such that the CCG attributes contained therein are accessible and identifiable by (i) HTML compliant editors and/or (ii) HTML compliant web
- 25 crawlers for the automatic construction of databases of classified information, and/or (iii) HTML compliant browsers for display on the computer screens of network users.

From another aspect, the invention comprises a method of constructing web pages of the above described type. The web pages may be constructed on digital computers using simple

30 text editors such as Microsoft Windows Notepad, or preferably, purpose built human controlled editors or automated composing programs which embody knowledge of HTML and CCG syntax and grammar. Which ever process is used, CCG attributes are selected and inserted, modified, deleted and/or organised to form a valid CCG phrases in HTML encoded documents and the documents are posted on computer readable storage devices of computers connected

35 to a computer network so that the documents are generally available to computers on the network.

From another aspect, the invention comprises a method of populating a database with CCG-data extracted from web pages. Web pages posted on a network are successively retrieved by

40 a digital computer program (eg: a web crawler) and CCG phrases contained therein are identified and at least some of the CCG attributes found within the CCG phrases are extracted. The CCG attribute names are used to determine the type of data in the associated values. Generally the CCG attributes of interest are those relating to classification, contact and geographic data and database update controls while the attributes of little or no of interest in

45 relation to database updating are those relating to display controls. Of course, the CCG-data extracted need only be that relevant to the particular database being updated. For example, one database may have been designed to index only web page classifications and URLs while another database may have been designed to index only contact details. Databases also differ in their internal representation of data and means of associating data. For example, some use

"flat file" tables, others use pointers to data to create network associations while others use hashing and buckets.

The conventional nomenclature differs considerably between different types of database. Depending on the particular database nomenclature, data of the same type is said to be stored in table columns, fields, attributes and properties. The terms column and field are somewhat related to the physical representation of the data in files while attribute and property is more related to the logical representation of data. To avoid confusion, with the terms "HTML attribute", "CCG attribute" or just "attribute", hereinafter a database property means both a type of data stored in the database and a place in the database where data of the same type is stored. Database properties are referred to by a name ("property name") or similar reference and contain values. For example, a database property with the name "City name" and which contains values which are all the names of cities may be defined as a "City name" type database property.

15
 20
 25
 30
 35
 40
 45
 50
 55
 60
 65
 70
 75
 80
 85
 90
 95
 100
 105
 110
 115
 120
 125
 130
 135
 140
 145
 150
 155
 160
 165
 170
 175
 180
 185
 190
 195
 200
 205
 210
 215
 220
 225
 230
 235
 240
 245
 250
 255
 260
 265
 270
 275
 280
 285
 290
 295
 300
 305
 310
 315
 320
 325
 330
 335
 340
 345
 350
 355
 360
 365
 370
 375
 380
 385
 390
 395
 400
 405
 410
 415
 420
 425
 430
 435
 440
 445
 450
 455
 460
 465
 470
 475
 480
 485
 490
 495
 500
 505
 510
 515
 520
 525
 530
 535
 540
 545
 550
 555
 560
 565
 570
 575
 580
 585
 590
 595
 600
 605
 610
 615
 620
 625
 630
 635
 640
 645
 650
 655
 660
 665
 670
 675
 680
 685
 690
 695
 700
 705
 710
 715
 720
 725
 730
 735
 740
 745
 750
 755
 760
 765
 770
 775
 780
 785
 790
 795
 800
 805
 810
 815
 820
 825
 830
 835
 840
 845
 850
 855
 860
 865
 870
 875
 880
 885
 890
 895
 900
 905
 910
 915
 920
 925
 930
 935
 940
 945
 950
 955
 960
 965
 970
 975
 980
 985
 990
 995

Whichever style of database is used, it is preferred that the database update program relate the CCG attributes to corresponding database properties used by the database update process so that the database property values are updated with CCG values in a manner which preserves the distinctness, content and meaning of the CCG values and, preferably, preserves the CCG value associations expressed in the CCG phrase as sets of associated database property values of different types.

In some cases, it is desired to know the address of the web page from which the CCG values were extracted. For example, the purpose of building a database might be to allow searching of the database by web page classification to provide a list URLs of web pages or URLs of portions of web pages which contain matching CCG classifications. The URLs could then be inserted in an HTML document and transmitted to a web browser as a list of references to web pages matching a search expression. In that example, associating the URL of a web page or the URL of a portion of a web page with the CCG values extracted from the same web page or web page portion is important and the URL or means of reconstructing it must be available and supplied to the database update process. In one style of database, the values of the same type are held separate rows in a column (property) of a database table, and pointers held in another column (property) are associated with the values by sharing the same table row. The table row constitutes a set of associated property values. Each pointer points to a bucket (block of data) containing a list of URLs or pointers to URLs held in a separate bucket or table. In another style of database, values of different types are held in different tables together with a set number, pointer or similar code which is used to indicate which values are associated as members of the same set. In one variation, the values of set members are prefixed with a code indicating the type of value and all values are held in the same column of a table. If the purpose of the database is to hold contact data, recording the web page URL in the database might not be required although if the URL is not present in the database, updating changes in the CCG contact details contained within a web page is more difficult. Of course, one database may be used to record all types of CCG values contained in web pages and associate with each other any and all values extracted from the same web page or even from other web pages.

From another aspect, the invention comprises a method of searching the databases constructed as outlined above. These databases may be used for a variety of searching purposes. For example, to find web page URLs by using the association of web page URLs with industry, service, product or subject classification or a person's or organisation's name or

address or geographic location values or any combination thereof. In another example, the databases may be used to find the contact details for people or organisations by name or location of industry, service, product or web page subject type and so forth by using the association between items of the contact details in the database without having to retrieve web pages associated with the contact details.

More particularly, the searching method involves finding URL references, or finding sets of associated database property values, from databases containing CCG-data. The method including steps of parsing a query phrase received from a computer network to extract query relational expressions and, from each expression, deriving a query field name, query relational operator and query value, determining the type of the query field by reference to its name, relating the query field to a corresponding database property according to type and locating CCG-data database property values in the database property which return a true value when tested against the query value using the query relational operator. Finally, the URL references or the sets of property values associated with the so located CCG-data database property values are extracted.

Database queries are usually expressed in a query language in the form of a phrase or sentence. In query by example style enquiry systems, the user types values into input fields on a form and a program extracts the input values and uses the values to automatically compose a query phrase or sentence. There are many existing examples of query languages used in connection with databases. Generally, they consist of relational expressions (eg Field=Value), logical expressions and grouping of relational and logical expressions by means such as parentheses. They may also contain sorting and output formatting expressions. Often abbreviated notation is used in the expressions such as leaving out field names or relational operators which are then inferred from the value in the expression or implied by default. In an enquiry the nature and format of the output may also be implied, such as a list of URLs of web pages or a list of contact details. Whatever is the mechanism of any particular database, the query expression needs to be parsed and fields in the query expression, explicit, default, implied or inferred, need be related to database properties of similar type. In some styles of database enquiry the query expression is evaluated against each row of a table or record of a file to find rows or records (ie a set of associated property values) which match the query expression. In other styles, sub-sets of the values of the properties are selected according to the interpretation of relational expressions in the query expression and the sub-sets are combined according to logical and grouping expressions in the query to find the sets of associated property values which match the query expression. Often, to make logical operations which combine the selected sub-sets more efficient, it is not the values which are selected but pointers to the values (eg Table name and table row) or unique keys (eg URLs or pointers to URLs) associated with the values. For example, the AND logical operator is often used to combine two lists so that only values or pointers or keys common to both lists are found in the combined list. Usually, the query produces a result list which is then provided to other processes. For example, a list of URLs of web pages is processed to produce an attractively formatted HTML encoded document containing the URLs and is sent to a web browser to allow an enquirer to retrieve interesting web pages. In another example, the contact details associated in the database with each value or pointer in the result list are retrieved from the database and presented as a report in the form of an HTML encoded document and is sent to a web browser for viewing.

From another aspect, the invention comprises a method of displaying CCG-data contained in CCG phrases within web pages which are displayed by a web browser executing on a digital

computer. While a web page is loading or has loaded in a web browser, the web browser parses the web page and displays the text (or data) of the web page on a display device connected to the computer. When the web browser parser encounters CCG phrases, the web browser may display the CCG-data (element and/or attribute names (or translations of element and/or attribute names) and/or values) in a number of browser specific ways. For example, the web browser may by default not display any CCG-data, display all CCG-data, not display any CCG-data until a CCG display control attribute explicitly states that subsequent data should be displayed or display all CCG-data until a CCG display control attribute explicitly states that subsequent data should not be displayed. The web browser may also use CGA display controls specifying the size, font, position and so forth to alter the display of the CCG-data.

DESCRIPTION OF EXAMPLES

Having indicated the nature of the present invention, examples or embodiments thereof will now be described by way of illustration only.

15

Example 1: HTML Syntax Suitable for Representing a CCG Phrase

The following is an example of HTML element syntax suitable for representing CCG phrases in which a control (e.g. "SHOW") may be "good until countermanded" and thus apply to more than one field:

```

20  <CCG HREF="url"
    {(NAME="label" | ID="identifier_code"} &| {LANG="language_code" &
    CLASS="Class_name"}
    {
25      {SET_SEPARATOR} &|
      {INDEX | NOINDEX} &|
      {SHOW | HIDE} &|
      {XPOS="horizontal_position_number"} &|
      {YPOS="vertical_position_number"} &|
      {NEWLINE} &|
30      {ALIGN=centre | left | right | justify} &|
      {SIZE=[+/-]1 | 2 | 3 | 4 | 5 | 6 | 7} &|
      {COLOR="#rrggbb" | "colour_name"} &|
      {FACE="type_face_name"} &|
      {BLINK &| BOLD &| UNDERLINE &| ITALIC &| STRIKE} &|
35      {SUBSCRIPT | SUPERScript} &|
      {CLEAR={left | right | all}}
      {NORMAL} &|
      {{{CONTACT &| COPYRIGHT &| DEVELOPER} &|
      {PERSONAL &| BUSINESS &| ASSOCIATION} &|
40      {attribute_name="attribute_value(s)}
    }
    ...
    >

```

where: the ellipsis "..." implies optional repetition of the braced ("{" ") items; the braces are used to group items and are not CCG syntactic elements; "&" (and) implies items must occur together, "|" (or) implies only one item must occur, and "&|" (and/or) implies any including none of the items may appear together.

Using the syntax of this example, each CCG phrase is represented as an HTML element, the element name being "CCG" and the CCG-data (eg attribute_name="attribute_value") and CCG

controls (eg SIZE=+1) are represented as attributes of the HTML element. Some of the attributes (eg SIZE) having explicit values (eg +1) and some attributes have implied values depending on the presence or absence in a CCG phrase (eg when the attribute BUSINESS is present it has the implied value of True and the implied value of False when absent).

5

Representation in XML syntax requires, at most, only a simple translation. All the items, such as "NORMAL" and "attribute_name" may remain unchanged as attributes of the element named "CCG" (eg <CCG size=+1/>). However, when a CCG phrase is encoded in XML, it is preferred that the items are represented as XML elements. For example attribute "SIZE=+1" can be represented as element "<size>+1</size>" or "<size value=+1/>" and "NORMAL" can be represented as "<normal/>".

In this example, the attributes, ID, LANG and CLASS take their meanings from HTML 3.0. The "url" in HREF="url" or may be a link with or without destination anchor labels. For example the URL <http://www.w3.org/docs.html> does not contain a destination anchor label (or identifier) while <http://www.w3.org/docs.html#searching> does contain the destination anchor label "#searching" which is intended refer to an anchor in docs.html such as There is some confusion in various HTML standards documentation about the distinction between the expression NAME="label" and the expression ID="identifier_code". For most practical purposes the two expressions have the same function or meaning: to uniquely identify within a document a position in or portion of that document.

Database control attributes:

"Set_separator" indicates the end of association between preceding and following data other than through the weaker mutual association with the same CCG phrase or web page; the data are divided into sets. "Index | Noindex" indicates that the following data are / are not to be indexed by a web crawler. These attributes have an implied attribute value of 'True' if present in and 'False' when absent from a CCG phrase.

30 Display control attributes:

"Show | Hide" indicates that a browser should show / not show the following data. Xpos and Ypos indicate the position (for example in pixel or physical units) on the browser screen where the data is to be displayed. "Newline" may be used in addition or as an alternative method of placing text on a browser screen. "Align" indicates the positioning of data on a browser screen relative to the cursor position set by "Xpos", "Ypos" or "Newline". "Size", "Colour" and "Face" indicates the size, colour and type face or font of the following data when displayed on an browser screen. "Blink", "Bold", "Underline", "Italic", "Strike", "Superscript" and "Subscript" indicates that the following data should be displayed blinking, bold, underlined, italicised, struck through, superscripted or subscripted. "Clear" indicates that the browser screen in the region where data will be displayed should be cleared to background before displaying the following data. "Normal" indicates the data is to be displayed without the "Blink", ..., "Clear" characteristics. The display controls which consist of an attribute name without an explicit value have an implied value of 'True' when present and 'False' when absent.

45 CCG-data attributes:

"Contact &| Copyright &| Developer" indicates that the following CCG-data refers to details for a person or organisation and/or to the copyright owner and/or to the HTML or web page developer. "Personal &| Business &| Association" indicates that the following data refers to details for a person and/or business and/or association. The previous CCG-data attributes have an implied attribute value of 'True' if present in a CCG phrase or set and 'False' when

absent from a CCG phrase or set. The attribute_name could be standard CCG attribute names or synonyms of standard CCG attribute names or abbreviations of CCG attribute names which refer to the following types of CCG attribute values where square brackets "[" and "]" surround suggested attribute names:

- 5 • industry or service or product or subject classifications and sub-classifications:
 - classification name [CN],
 - classification codes [CC].
- display only text [TEXT].
- contact:
 - 10 • person:
 - courtesy title [PNC],
 - first given name [PNG],
 - other given names [PNO],
 - family name [PNF],
 - 15 • name suffix [PNS],
 - qualifications [PQ],
 - associations [PA],
 - contact person title [P-T],
 - contact person role [PR].
 - 20 • organisation:
 - name [ON],
 - unit [OU],
 - identifier [OID].
 - physical or post or delivery address:
 - 25 • type [AT] (= "PHYSICAL" &| "POST-OFFICE" &| "POSTAL" &| "DELIVERY")
 - post office box number [AP#]
 - post office name [APN]
 - room or suite or office or unit or flat or apartment name &| number [AB#],
 - floor name &| number [ABF],
 - 30 • building name [ABN],
 - lane or street or road or highway number [AS#],
 - lane or street or road or highway name [ASN],
 - suburb or town or city name [ACN],
 - region or state or territory or province name [ARN],
 - 35 • post code [APC],
 - country or nation name [ANN],
 - telephone:
 - type [TT] (= "PREFERRED" &| "VOICE" &| "MOBILE" &| "CAR" &| "MESSAGE" &| "PAGER" &| "FACSIMILE" &| "MODEM" &| "ISDN" &| "VIDEO")
 - 40 • nation or country code number [TC#],
 - trunk access number [TT#],
 - area code number [TA#],
 - local number [TL#],
 - email:
 - 45 • type [ET] (= "INTERNET" | {other}),
 - mailer [EM],
 - address [EA],
 - Internet address:
 - url [IURL].
 - 50 • date & time:

- date & time from [DTF].
- date & time to [DTT].
- weekday from [DTWF].
- weekday to [DTWT].
- 5 • weekday time from [DTWFT].
- weekday time to [DTWTT].
- time zone [DTZ].
- brand name [BN].
- public key:
 - 10 • key type [KT].
 - key [K].
- geographical:
 - location units [GLU].
 - location [GL].
 - 15 • serviced region units [GLRU].
 - serviced region [GLR].

Suggested attribute name [CN] is the name of an attribute associated with the attribute value containing "classification name" type data. For example, the [CN] attribute value could be the
 20 name of a proprietary or national or international or other industry classification standard such as the Australian and New Zealand Standard Industry Classification or "ANZSIC" for short or the U.S. Bureau of the Census Industrial Classifications (USBCIC). The associated classification codes [CC] attribute value could contain the codes and/or descriptions of the codes of the named standard with or without modifications, deletions or extensions. For
 25 example: CN="ANZSIC" CC="61;Road transport" or CN="USBCIC" CC="581;Hardware store". Service classifications such as the International Standard Classification of Occupations could be used. For example: CN="ISCOO" CC="4430;Auctioneer" Product classifications such as the Harmonised Commodity Description And Coding System could be used. For example: CN="HSC" CC="8411;Turbojets, turbo-propellers & other gas turbines; parts thereof" For
 30 subject classifications, Dewey Decimal, and/or Universal Decimal and/or Library of Congress and/or Bliss and/or Colon Classification could be used. For example: CN="DDC" CC="577.699;Sea shore ecology" The inclusion of subject classifications provides a very simple, straightforward method of classifying the subject matter of an HTML document which could be attractive to commercially oriented copyright owners.

35 The text ([TEXT]), person ([PNC] - [PR]), organisation ([ON] - [OID]), physical or post or delivery address ([AT] - [ANN]), telephone ([TT] - [TL#]), email address ([ET] - [EA]) and Internet address ([URL]) are intended to be associated with each other in the obvious manner. Date & time(s) ([DTF] - [DTZ]) are intended to indicate the times at which the address and/or
 40 telephone and/or email will be serviced by the associated person(s) and/or organisation(s). The brand name ([BN]) attribute is intended to hold commercial brand names. Public key ([KT] - [K]) is intended to hold public encryption keys for secure communication with the contact person or organisation.

45 The geographical location [GL] could be a latitude and longitude (eg E148D31'12.5",S36D40'09.6" or E148.5201,S36.6693 or -148.5201,-36.6693), or a Universal Grid Reference (eg 55FV364402) or other global, national, regional or local location reference with units as specified [GLU], which is typed in or obtained by pointing to a digitally encoded map or other methods. In more populated regions of some countries such as the U.S., street
 50 addresses and post codes are associated with a moderately accurate geographic location and

can be used to interpolate geographic location data where geographic location data is not explicitly stated in the CCG-data. Using a universally recognised code such as latitude and longitude has advantages when used with international mediums like the Internet. Geographical location is intended to be associated with a post, delivery address or physical address such as place of business or residence. A CCG compliant browser could use this reference to display a map centred on that geographic location. The purpose of the geographical location data is to allow browser users to specify search engine search criteria which will result in the search engine selecting only those Internet accessible documents which provide details about providers which are within a specified region. The serviced region [GLR] is intended to indicate the preferred area of operation of providers expressed in terms of serviced region units [GLRU]. A radial distance (eg in kilometres) or alternate means of expressing an area of interest around a geographic point, such as polygons, are envisaged.

It is envisaged that the CCG attribute_value could be composed of more than one value (actually sub-value) wherein specific characters or character strings separate individual values.

While specific instances of element names and types have been given in this example, of more importance is the type of data and type controls over the display and indexing of the data. As an alternative to the preferred immediately following example where the CCG-data is lumped together under the HTML element named "CCG", certain elements of the data, for example the classification data, could be lumped under separate HTML elements with distinctly different names thereby separating CCG classification data from CCG contact data. However, this is not preferred because the strength of association between the two types of data is weakened.

25

Example 2: Classification of Portion of a Web Page.

Where it is desired to classify a portion of a web page, such as a paragraph about a product, simple CCG-data may be used in conjunction with the syntax of Example 1. For example:

```

30      <A NAME="Radios">AM-FM radio receivers: </A>
          <CCG HREF="#Radios">
              CN="ANZSIC"
              CC="E23.34.78:Electrical equipment - radio receivers AM"
              CC="E23.34.79:Electrical equipment - radio receivers FM"
          </CCG>

```

35

We won't be beaten on the price of these high quality receivers

In this example, the CCG phrase appears after the related anchor (<A NAME=...). However, while such proximity visually provides an obvious association between the anchor and related CCG phrase, it is intended that CCG phrase containing the attribute HREF related to a specific anchor could appear anywhere within the body of a web page and remain related to the named anchor. The CCG phrase containing the attribute HREF could appear in a separate document and thereby relate the CCG-data to the entire document or to a named anchor although, as previously noted, coordinating separate documents can be problematic. In the absence of the HREF and NAME attributes, it is also intended that the CCG-data apply to the whole web page.

45

Example 3 Classification of Portion of a Web Page using XML Syntax

Using XML syntax and similar attribute names to those of Example 2 the HTML fragment of Example 2 may be rewritten as:

```

50      <A NAME="Radios">AM-FM radio receivers: </A>
          <XML>

```

```

5      <CCG>
        <HREF>"#Radios"</HREF>
        <CN>"ANZSIC"</CN>
        <CC>"E23.34.78;Electrical equipment - radio receivers AM"</CC>
        <CC>"E23.34.79;Electrical equipment - radio receivers FM"</CC>
      </CCG>
    </XML>

```

We won't be beaten on the price of these high quality receivers

This example demonstrates that the translation of CCG-data from HTML to XML (and the reverse) involves simple syntactical and grammatical translations. Of course, the resulting HTML and XML, while "well formed" might not be recognised or, if recognised, might not be understood by some parsers.

Example 4: Constructing a Web Page Containing CCG-data

- 15 As an example, a web page developer, Alice Jamieson, is preparing an advertisement for a local electrician John Williams, trading as Kelso Electrical, who wants to advertise on the web for business within 30 kilometres from his office located at 18 Raglan Street, Kelso, New South Wales. Alice uses a graphical user interface web page authoring tool capable of creating and modifying web pages containing HTML (and XML) CCG phrases by accepting inputs from a
- 20 user. The tool executes on a digital computer having input devices such as a keyboard, mouse, light pen and touch pad, display devices such as a CRT, LED arrays, liquid crystal arrays and computer-readable media such as magnetic and optical disks, memory arrays, magnetic tape and the like.
- 25 The authoring tool also embodies knowledge of the content and structure of CCG phrases such as the attribute names, valid ranges and sets of associated attribute values, the normal order of the attributes in the CCG phrase and interdependencies between attribute values. The tool provides a window where web pages may be viewed in layout (browser) mode and another window where the HTML code may be viewed in editing mode. The tool also provides
- 30 means of inserting, deleting, modifying and organising HTML elements, changing font size, face and colour and so forth. The tool provides means for the user to build CCG phrases by using input devices to select an edit control representing various types of CCG attributes from a list which the tool then inserts in the body of a web page together with, when not already present, HTML code indicative of the start and end of a CCG phrase. The user then types in
- 35 the value in the attribute. Similarly, the tool provides means of converting web page text to CCG attributes. Using input devices, the user selects the text to be converted to a CCG attribute then selects an edit control from a list; the tool then inserts the HTML code necessary to encode the text as a CCG attribute. However, these semi-manual methods of creating and modifying CCG phrases are inefficient and error prone. The tool also provides a button, which
- 40 can be activated by using input devices, for access to CCG phrase editing functions. The CCG editing functions consist of a means of extracting the CCG values from existing CCG phrases in the web page being edited, forms for entering and modifying the extracted CCG values, a layout view browser window for altering how the CCG-data displays (position, font size, face, colour, bold, normal, hiding or showing and so forth), a data view browser window to alter
- 45 which CCG-data values are to be indexed or not indexed in search engine databases, and a means of deleting existing CCG phrases from web pages and inserting new or changed CCG phrases in web pages. Editing cursors marking the current location at which text and/or data may be inserted, deleted or modified are provided in each window and form.

In the current example, the web page initially contains no CCG phrase. Clicking the CCG editing function button of the authoring tool causes a form to appear. The form contains prompts related to CCG attribute names and associated data input fields related to the CCG attribute values associated with the CCG attribute names, that is CCG-data. The fields are blank because, in the web page layout view, the edit cursor is not over a CCG phrase (and can not be since the web page initially contains no CCG phrase). The service classifications relevant to the web age, John Williams physical business contact address, phone and fax numbers, email address and geographic location and his post office business contact addresses are entered into the forms using a keyboard and mouse. The developer, Alice Jamieson, also includes her basic contact details where provided for on the form. The forms use drop down lists to select address blocks (eg physical and post office) for editing. Logic associated with the forms validates the CCG attribute values and interdependencies. Input devices are then used to control the CCG-data layout view browser to modify the appearance of the CCG-data such as font size and colour and positioning. In the layout browser, input devices communicating with the edit cursor are used to highlight individual items and blocks of items to be changed. The post office address is highlighted as a block and moved into position in line with the physical address. The CCG-data view window is then used to check which data items are to be indexed by search engines. In this example all CCG-data (ie all CCG attribute values except display control values and database control values) are to be indexed. Input devices are used to control the edit cursor to highlight the entire data and a mouse is used to click (activate) a button to mark all the data for indexing. Then another button is clicked which builds an HML encoded CCG phrase of CCG attributes derived from the CCG-data values, display control values and database control values and inserts the CCG phrase in the web page at the location pointed to in the web page layout browser window.

The HTML code editing mode window was called up which revealed the following HTML encoded CCG phrase in the web page:

```

    <XML>
    <CCG>
30      <INDEX/>
        <HIDE/>
        <CN>ANZSIC</CN>
        <CC>D36.11.45;Electrical contractors - residential</CC>
        <CC>D36.11.46;Electrical contractors - industrial</CC>
35      <SHOW/>
        <CONTACT/> <COPYRIGHT/>
        <BUSINESS/>
        <XPOS>50</XPOS>
        <YPOS>320</YPOS>
40      <ALIGN>centre</ALIGN>
        <SIZE>3</SIZE>
        <COLOR>black</COLOR>
        <FACE>Times New Roman</FACE>
        <BOLD/>
45      <CLEAR>all</CLEAR>
        <TEXT>Contact :</TEXT>
        <PNC>Mr</PNC>
        <PNG>John</PNG>
        <PNF>Williams</PNF>
50      <PQ>AIE</PQ>

```

<PA>ARUC</PA>
 <NEWLINE/>
 <PT>Managing Director</PT>
 <NEWLINE/>
 5 <ON>Kelso Electrical Pty. Ltd.</ON>
 <NEWLINE/>
 <NORMAL/> <ITALIC/>
 <SIZE>-2</SIZE>
 <TEXT>NSW License 45678C</TEXT>
 10 <NEWLINE/>
 <NORMAL/> <BOLD/>
 <SIZE>+2</SIZE>
 <AT>PHYSICAL</AT>
 <AS#>18<AS#>
 15 <ASN>Raglan Street<ASN>
 <NEWLINE/>
 <ACN>Kelso</CAN>
 <NEWLINE/>
 <ARN>NSW<ARN>
 20 <NEWLINE/>
 <HIDE/>
 <ANN>Australia</ANN>
 <NEWLINE/>
 <SHOW/>
 25 <TEXT>Phone:</TEXT>
 <TT>PREFERRED ; VOICE ; MESSAGE</TT>
 <HIDE/>
 <TC#>61</TC>
 <SHOW/>
 30 <TT#>0</TT#>
 <TA#>63</TA#>
 <TL#>456-7828</TL#>
 <TEXT> Fax:</TEXT>
 <TT>FACSIMILE</TT>
 35 <HIDE/>
 <TC#>61</TC#>
 <SHOW/>
 <TT#>0</TT#>
 <TA#>63</TA#>
 40 <TL#>456-7829</TL#>
 <NEWLINE/>
 <ET>INTERNET</ET>
 <EA>johnw@firefly.com.au<EA>
 <TEXT> </TEXT>
 45 <GLU>LatLong</GLU>
 <GL>="33.3978S;148.5679E</GL>
 <GLRU>Km</GLRU>
 <GLR>30 </GLR>
 <SET_SEPARATOR/>
 50 <XPOS>250</XPOS>

5 <YPOS>320</YPOS>
 <NEWLINE/>
 <NEWLINE/>
 <TEXT>Or write to us at :</TEXT>
 <NEWLINE/>
 <ON>Kelso Electrical Pty. Ltd.</ON>
 <NEWLINE/>
 <AT>POST-OFFICE</AT>
 10 <AP#>P.O. Box 187</AP#>
 <NEWLINE/>
 <APN>Sunny Corner</APN>
 <TEXT> </TEXT>
 <APC>2795</APC>
 <NEWLINE/>
 15 <HIDE/>
 <ANN>Australia</ANN>
 <SET_SEPARATOR/>
 <HIDE/>
 <DEVELOPER/>
 20 <BUSINESS/>
 <PNG>Alice</PNG>
 <PNF>Jamieson</PNF>
 <ET>INTERNET</ET>
 <EA>alijam@firefly.com.au</EA>
 25 <IURL>http://www.firefly.com.au/~aljam/</IURL>
 </CCG>
 </XML>

In the web page layout browser window the CCG-data displayed as follows:

30	Contact : Mr John Williams, AIE, ARUC, Managing Director Kelso Electrical Pty. Ltd. NSW License 45678C 18 Raglan Street Kelso NSW Phone: 063-456-7828 Fax: 063-456-7829 Email: johnw@firefly.com.au <u>Map</u>	Or write to us at: Kelso Electrical Pty Ltd P.O. Box 187 Sunny Corner 2795
35		
40		

Having encoded the web page in this way, Alice then posts it on the storage device of a digital computer connected to the Internet from where it can be retrieved through the Internet using the URL "http://www.firefly.com.au/~johnw/index.html"

45 Example 4: Constructing a Database from Web Pages Containing CCG-data

During a routine sweep of Internet connected web page servers, a web crawler (or robot) operating on a server named "ccg.search.com" executing on an Internet connected digital computer discovers the URL "http://www.firefly.com.au/~johnw/index.html" in a document it had previously retrieved through the Internet. The web crawler decides that the URL matches 50 its selection criteria because the URL contains the suffix ".html". The web crawler then

successfully retrieves the document by extracting from the URL the address of the computer hosting the document, addressing and sending a message (including the address of the web crawler) requesting the web page through the network to the web page host computer using TCP/IP protocol, the host computer then reads the document, addresses and sends the document to the web crawler using TCP/IP protocol, the web crawler then waiting until it has received all parts of the web page from the host computer before proceeding. It inspects the contents of the document and finds that it matches the additional selection criteria that it is an HTML encoded document. The web crawler program, depending on its state and logic, then parses the document, strips out and saves some or all of the URLs in the document for future examination. The web crawler program then passes the document, together with the URL of the document through a network communications channel to an indexing program executing on a different computer. The indexing computer has database updating software which manipulates a database stored on computer-readable media.

- 15 The indexing program parses the document, from first to last character, indexing some of the meta data in the <head> of the document and the words in the text of the document with respect to the document URL. In the database of this example, unique words extracted from the documents already indexed are held in separate rows of a column of a database table and in another column of the same table on each row is an associated pointer to the first bucket or block of URLs of documents containing the word associated with the pointer. As new words are found, the new word is added as a new row in the word column of the table, a new bucket is created, the URL of the document containing the new word is inserted into the bucket and a pointer to the new bucket is written in the new row pointer column. When the same word is found in another document, the row in the table of the word is found, the pointer is retrieved from the table, the bucket pointed to by the pointer is retrieved and the URL of the other document is inserted in the bucket. Where a bucket becomes full of URLs, a new bucket is created and a pointer to the new bucket for holding additional URLs is placed in the full bucket. Deletion of words and URLs of changed or no longer existing documents is also provided for.
- 30 In addition to indexing words extracted from the text of the document, the indexing program also indexes the CCG-data in the document as well as indexing words found in the CCG-data. When the parser finds HTML element "<XML>" in the document it switches into XML parsing mode and switches out of that mode when "</XML>" is found. When the element "<CCG>" is found, the parser switches into the CCG parsing mode and switches out of that mode when "</CCG>" is found.

The example database has a CCG-data attribute name to database property name correspondence table to show the relationship between the CCG-data attribute names and the database tables and columns (properties) where the CCG-data attribute values are to be stored in the database as database property values. The database property values and associated URLs are stored in much the same way as for words extracted from text as outlined above. However, CCG contact data, for example, which consists of several distinct CCG-data attributes which are related (eg street name, city), is stored in a database table having a column (property) related to each distinct CCG contact attribute name and each separate CCG contact data set (eg person's name, address, telephone number) as separated by "<CCG>", "<SET_SEPARATOR>" and "</CCG>" is held in a separate row in the table. The values stored in each row are considered to be a set of associated property values of different types.

The indexing program, during parsing the document of Example 2 above, encounters the "<CCG>" element and enters the CCG parsing mode. The parser knows to ignore display control attributes and to consider database control elements in the CCG phrase. The example indexing program opts to index all other CCG-data contained in the attribute values until explicitly instructed not to index the attribute values by encountering the "<NOINDEX/>" database control element and then to recommence indexing when the "<INDEX/>" database control element is encountered.

- Taking each CCG-data attribute name and associated attribute value(s) in succession, the example indexing program uses the correspondence table to translate the CCG-data attribute name to the database table and column (property) names where the CCG-data attribute value(s) are to be stored as database property value(s). The indexing program may opt to translate the CCG-data attribute values to database property values by, for example, converting character strings of digits to binary encoded decimal representation, the string "True" to a single bit representation and the like. The indexing program then adds or updates the database property value(s), using the database table and column (property) names (or similar references) obtained by translation, in much the same manner as outlined above for the update of the database using words extracted from the document text, including associating the data to the document URL where desired. Where the CCG-data contains a "HREF" attribute (or similar), the URL associated with the other CCG-data is a URL taken from the "HREF" attribute value or composed of the document URL and the "HREF" attribute value if the attribute value is a partial or relative URL. Some CCG attributes, such as "<BUSINESS/>" have only an implied value of true if the attribute is present and false if the attribute is absent, the "<SET_SEPARATOR/>", "<CCG>" and "</CCG>" resetting such values to false. However, where attribute value(s) associated with different attribute names are still related, such as a person's name and a street name, the related values of different types are stored on the same row of the same database table but in a different column (database property) to preserve the relationship. "<SET_SEPARATOR/>" limits the degree of relatedness between, for example, a person's name occurring before the separator and a street name occurring after the separator.
- Using the example document and using the same database column (property) names as used for the CCG-data attribute names a portion of the table constructed database table would look like:

	PNC	PNG	PNF	PQ	PA	PT		URL
...
...	Mr	John	Williams	AIE	ARUC	Managing Director	...	(pointer)
...

- Difficulties not highlighted by this example are the need to handle properties having multiple values of the same type, "sparse rows" where only a few values are not null (blank) and tables with extremely large numbers of rows. For example, the CCG-data of this example could have contained multiple values of personal qualifications ("PQ"). To represent this type of data using a 2 dimensional table database system, the database would be "normalised" so that the multiple values were stored in a separate table and keys or pointers were used to relate the items in the two tables. Numerous alternate database systems, for example those based on key hashing and data buckets, or tagging data values with prefixes or suffixes related to the type of data value may be used. Preferably, however, whatever database system is used, it should preserve the associations of CCG-data items present in the CCG phrases.

Because the geographic location data was missing from the postal address of the CCG-data in the example document, but a post code was present, the indexing program inferred the geographic location from the post code.

5

Example 6: Finding Web Page References Using a CCG Database

As an example, Kevin Robson lives in Sydney but owns and has rented out a house in Bathurst. He wants to use the web to find some electricians based in the general Bathurst region (not only in Bathurst City) to contact for estimating the cost of modifying the wiring in the house. He uses his web browser to open the web page ["http://www.ausline.com.au/web_search.html"](http://www.ausline.com.au/web_search.html) containing AusLine's search engine web page search criteria input form encoded using the HTML "<form>" element.

The search criteria input form contains several input fields including those labelled "Service classification", "Key words", "City./Suburb/Town", "Country", "Lat/Long" and "Radius". The form also displays a button labelled "Map" to allow latitude and longitude to be selected by pointing to map images. The word "electrician" is typed into the "Service classification" field, "house wiring" into the "Keywords" field, "Bathurst" into the "City/Suburb/Town" field and "10" into the field "Radius". The country "Australia" was already showing in the country field because the web page server had received cookie data from the browser indicating that that was the country used when the browser last used the web page. The "submit search" button on the web page was clicked. The browser transmitted a message using TCP/IP protocol to the AusLine server containing the input field values encoded in the header of the message.

After a short delay, the search result HTML encoded web page was returned. Clicking on the "Service classification" input field drop down list box to check the classifications used in the search revealed three items:

- Electrical contractors - residential
- Electrical contractors - industrial
- Electrical engineers

The search engine attached to the server obtained those classifications by using word stemming and searching the text of the service classifications held in it's database. The Lat/Long field contained the value "33.3856S;148.5743E" which the search engine obtained by looking up the latitude and longitude of the town "Bathurst" in the country "Australia" in it's database. Clicking on the "Map" button retrieved a web page having the image of a map centred on the town of Bathurst and showing the area 20 Km around it. The search engine obtained the map by making a request to another Internet connected server and supplying the latitude, longitude and radius. Clicking on the browser "Back" button returned to the search results page.

40

The search results contained 8 titles, brief descriptions and URLs including a reference containing the URL ["http://www.firefly.com.au/~johnnw/index.html"](http://www.firefly.com.au/~johnnw/index.html). Retrieving each in turn revealed that all were well focused according to the search criteria being related to electricians, electrical contractors and engineers in the Bathurst area. The search engine obtained these references to web pages by:

- searching it's database of service classification titles with words stemming from "electrician" which resulted in three service classification codes,
- searching it's database using the three service classification codes to obtain an intermediate list of URLs of web pages containing those CCG codes

- searching it's database for the two keywords to obtain an intermediate list of URLs of web pages containing those words in the web page text,
- Searching it's database to find the latitude and longitude of Bathurst, Australia,
- 5 • searching it's database to obtain an intermediate list of web pages which contain latitude and longitude data lying within 10 Km of the latitude and longitude of Bathurst, Australia,
- producing as a result list, a list of URLs which are common to all the intermediate lists,
- obtaining from it's database the title and brief description of the web pages,
- 10 • formatting the titles, descriptions and URLs into an HTML encoded report,
- transmitting the report to the enquiring web browser.

Example 7: Finding Contact Details Using a CCG Database

As an example, Jim Jones of Jones and Sons wants to send a recall notice about a faulty batch of UV stabilised electrical power cable to all Electrical contractors and Electrical
15 wholesalers in Australia who have email addresses. He uses his web browser to open the web page "http://www.ausline.com.au/contact_search.html" containing AusLine's search engine contact search criteria input form encoded using the HTML "<form>" element.

The search criteria input form contains several input fields including those labelled "Service
20 classification", "Country" and "Output format". The word "electric" is typed into the "Service classification" field, the word "Australia" is typed into the "Country" field and the "Tabular - Name & Email" option in the "Output format" drop down list box is selected. The "Submit search" button on the web page is clicked. The browser transmits a message using TCP/IP protocol to the AusLine server containing the input field values encoded in the header of the
25 message.

After a short delay, the search result HTML encoded web page is returned. Clicking on the "Service classification" input field drop down list box to check the classifications used in the search revealed too many classifications for the result to be sufficiently focused. The following
30 four classifications were selected from the list:

- Electric cable - ducting systems
- Electrical contractors - residential
- Electrical contractors - industrial
- Electrical wholesalers

35 and the "Submit search" button is pressed again to refine the search.

The search results contained 3,473 names and associated email addresses and URLs to full contact details. Jim saved the search result page on his computer so that he could use his email program to send the recall notice to each email address in the list. The email address
40 "johnw@firefly.com.au" was included in the list.

The search engine obtained these references to web pages by:

- searching it's database using the four service classification titles which resulted in four service classification codes,
- 45 • searching it's database using the four service classification codes to obtain an intermediate list of database primary keys of database table rows containing those service classification codes in the database Service classification attribute,
- searching it's database using the country name "Australia" to obtain an intermediate list of database primary keys of database table rows containing that word in the
50 database Country attribute,

- producing as a result list, a list of database primary keys which are common to both the intermediate lists,
- obtaining from it's database using the result list the values of the name and email attributes,
- 5 • using the HTML <table> element to format the name values, email values and full detail URLs into an HTML encoded report,
- transmitting the report to the enquiring web browser.

10 This example relates to finding sets of associated database contact values without requiring references to web pages. However, finding other sets of associated database values such as sets of associated industry classification values and geographic location values might also be useful for some purposes.

15 Thus it is appreciated that the afore stated goals, advantages and objectives are achieved by the teachings herein. In particular it is seen that, unlike the prior art, efficiently searchable Yellow pages and White pages databases and the like may be automatically constructed from HTML encoded web pages. Additionally the database entries may be automatically linked to specific web pages and portions of web pages allowing convenient methods of indexing of product and service catalogues and the like. It is also appreciated that simpler methods of
20 constructing databases suited to a variety of other uses such as industry and subject directories are also provided.

25 From the foregoing teachings and with the knowledge of those skilled in the art, it is apparent that other modifications and adaptations of the invention will become apparent. For example, the method steps disclosed and claimed herein may be practiced in a variety of different orders. CCG-data may take on a variety of different forms within the meaning of the claims. Thus, it is our intention to include within the scope of the claims not only the invention literally embraced by the language of the claims but to include all such modifications and adaptations which may come to those skilled in the art.

What I claim is:

1. An HTML encoded web page embodied on a computer-readable medium, said web page comprising at least one HTML encoded CCG phrase, each CCG phrase comprising:
 - a) HTML code indicative of the start of a CCG phrase,
 - b) at least one CCG-data attribute, and
 - c) HTML code indicative of the end of a CCG phrase.
2. An HTML encoded web page embodied on a computer-readable medium, said web page comprising at least one HTML encoded CCG phrase, each CCG phrase comprising:
 - a) HTML code indicative of the start of a CCG phrase,
 - b) at least two CCG-data attributes,
 - c) at least one database control attribute separating said CCG-data attributes into at least two sets of CCG attributes, and
 - d) HTML code indicative of the end of a CCG phrase.
3. An HTML encoded web page embodied on a computer-readable medium, said web page comprising at least one HTML encoded CCG phrase, each CCG phrase comprising:
 - a) HTML code indicative of the start of a CCG phrase,
 - b) at least one CCG-data attributes,
 - c) at least one attribute of: database control attributes, display control attributes; and
 - d) HTML code indicative of the end of a CCG phrase.
4. A computer implemented method of building a web page comprising at least one HTML encoded CCG phrase, the method comprising the steps of:
 - a) displaying a web page on a computer display device,
 - b) displaying an edit cursor indicating a character position on said display device and a corresponding character position in said web page, said edit cursor being positionable within the display of said web page by use of computer input devices,
 - c) separately displaying on said computer display device a set of edit controls representing CCG-data attribute types,
 - d) positioning said edit cursor within said display of said web page using said input devices,
 - e) selecting an edit control from said set of edit controls using said input devices,
 - f) relating said selected edit control to a corresponding CCG-data attribute name,
 - g) constructing a CCG-data attribute character string comprising a character string representing said attribute name and another character string representing an empty CCG-data value,
 - h) if the said edit cursor is positioned outside a CCG phrase,
 - i) inserting into said web page, at the character position indicated by said edit cursor, a start character string comprising HTML code indicative of the start of a CCG phrase,
 - ii) inserting into said web page, immediately after the end of said start character string, an end character string comprising HTML code indicative of the end of a CCG phrase, and
 - iii) positioning said edit cursor between said start and end character strings,

- i) inserting said CCG-data attribute character string into said web page at the character position indicated by said edit cursor,
 - j) positioning said edit cursor at the character position in said web page of the CCG-data value of said inserted CCG-data attribute character string,
 - 5 k) inputting characters using a keyboard,
 - l) inserting said input characters into said web page at the character position indicated by said edit cursor, thereby converting said empty CCG-data value to a non-empty CCG-data value, and
 - 10 m) writing said web page on computer-readable media.
5. A computer implemented method of building a web page comprising at least one HTML encoded CCG phrase, the method comprising the steps of:
- a) displaying a web page on a computer display device,
 - 15 b) displaying a start edit cursor and an end edit cursor on said display device, each said edit cursors indicating a character position on said display device and a corresponding character position in said web page, said edit cursors being positionable within the display of said web page by use of computer input devices,
 - c) separately displaying on said computer display device a set of edit controls representing CCG-data attribute types,
 - 20 d) selecting a string of web page characters on said display device using said input devices to position said start edit cursor to indicate the start said string of web page characters and said end edit cursor to indicate the end of said string of web page characters,
 - e) selecting an edit control from said set of edit controls using said input devices,
 - 25 f) relating said selected CCG-data control to a corresponding CCG-data attribute name,
 - g) constructing a CCG-data attribute character string comprising a character string representing said attribute name and another character string representing a CCG-data value containing said string of web page characters,
 - 30 h) deleting said string of web page characters from said web page,
 - i) if the said start edit cursor is positioned outside a CCG phrase,
 - i) inserting into said web page, at the character position indicated by said start edit cursor, a start character string comprising HTML code indicative of the start of a CCG phrase,
 - 35 ii) inserting into said web page, immediately after the end of said start character string, an end character string comprising HTML code indicative of the end of a CCG phrase, and
 - iii) positioning said start edit cursor between said start and end character strings,
 - 40 j) inserting said CCG-data attribute character string into said web page at the character position indicated by said start edit cursor, thereby converting said string of web page characters to a CCG-data attribute value contained within a CCG-data attribute contained within CCG-phrase, and
 - 45 k) writing said web page on computer-readable media.
6. A computer implemented method of building a web page comprising at least one HTML encoded CCG phrase, the method comprising the steps of:
- a) displaying a CCG-data input form on a computer display device,
 - 50 b) inputting CCG-data values into fields of said data input form using computer input devices,



(12) PATENT ABSTRACT (11) Document No. AU-A-53031/98
(19) AUSTRALIAN PATENT OFFICE

(54) Title
NETWORK-BASED CLASSIFIED INFORMATION SYSTEMS

International Patent Classification(s)
(51)⁶ G06F 017/30

(21) Application No. : 53031/98 (22) Application Date : 10/02/98

(30) Priority Data

(31) Number	(32) Date	(33) Country
PO5254	21/02/97	AU AUSTRALIA

(43) Publication Date : 27/08/98

(71) Applicant(s)
DUDLEY JOHN MILLS

(72) Inventor(s)
DUDLEY JOHN MILLS

(57)

A system for automatically creating databases containing industry, service, product and subject classification data, contact data, geographic location data (CCG-data) and links to web pages from HTML, XML or SGML encoded web pages posted on computer networks such as the Internet or Intranets. The web pages containing HTML, XML or SGML encoded CCG-data, database update controls and web browser display controls are created and modified by using simple text editors, HTML, XML or SGML editors or purpose built editors. The CCG databases may be searched for references (URLs) to web pages by use of enquiries which reference one or more of the items of the CCG-data. Alternatively, enquiries referencing the CCG-data in the databases may supply contact data without web page references. Data duplication and coordination is reduced by including in the web page CCG-data display controls which are used by web browsers to format for display the same data that is used to automatically update the databases.

- c) inserting into the body of a web page a start character string comprising HTML code indicative of the start of a CCG phrase,
 - d) inserting into said web page body immediately after the end of said start character string an end character string comprising HTML code indicative of the end of a CCG phrase,
 - e) extracting successive field values from said data entry form together with related field value type information,
 - f) relating the type of each extracted field value to a corresponding CCG-data attribute name,
 - g) constructing a CCG-data attribute character string comprising a character string representing said attribute name and another character string representing said field value,
 - h) inserting said CCG-data attribute character string into said web page between said start and end character strings.
 - i) writing said web page on computer-readable media.
7. A computer implemented method of building a database which comprises sets of associated property values wherein each set includes at least two property values of different types, the property values being any of classification values, contact values, geographic location values, hereinafter collectively referred to as CCG-data, the method comprising the steps of:
- a) retrieving successive web pages from a computer network, each web page being identified by a URL,
 - b) searching each web page for a CCG phrase that includes a plurality of different types of CCG-data attributes,
 - c) extracting a plurality of said attributes from said phrase,
 - d) from each extracted attribute, deriving an attribute name and a related attribute value,
 - e) determining the type of said extracted attribute and said attribute value by reference to said attribute name,
 - f) relating said type of attribute value so determined to a corresponding type of database property value,
 - g) relating the URL of said web page to an other type of database property value,
 - h) writing said derived attribute value to the database property value of said determined corresponding type in a set of associated property values, and
 - i) writing the URL of said web page to a database property value of said other type in said set of associated property values.
8. A computer implemented method of building a database which comprises sets of associated property values wherein each set includes at least two property values of different types, the property values being any of classification values, contact values, geographic location values, hereinafter collectively referred to as CCG-data, the method comprising the steps of:
- a) retrieving successive web pages from a computer network, each web page being identified by a URL,
 - b) searching each web page for a CCG phrase that includes at least one type of CCG-data attribute,
 - c) extracting at least one said attribute from said phrase,
 - d) from each extracted attribute, deriving an attribute name and a related attribute value,

- e) determining the type of said extracted attribute and said attribute value by reference to said attribute name,
 - f) relating said type of attribute value so determined to a corresponding type of database property value,
 - 5 g) relating the URL of said web page to an other type of database property value,
 - h) writing said derived attribute value to the database property value of said determined corresponding type in a set of associated property values, and
 - i) writing the URL of said web page to a database property value of said other type in said set of associated property values.
- 10 9. A computer implemented method of building a database which comprises sets of associated property values wherein each set includes at least two property values of different types, the property values being any of classification values, contact values, geographic location values, hereinafter collectively referred to as CCG-data, the method comprising the steps of:
- 15 a) retrieving successive web pages from a computer network,
 - b) searching each web page for a CCG phrase that includes a plurality of different types of CCG-data attributes.
 - c) extracting a plurality of said attributes from said phrase,
 - 20 d) from each extracted attribute, deriving an attribute name and a related attribute value,
 - e) determining the type of said extracted attribute and said attribute value by reference to said attribute name,
 - f) relating said type of attribute value so determined to a corresponding type of database property value, and
 - 25 g) writing said derived attribute value to the database property value of said determined corresponding type in a set of associated property values.
- 30 10. A computer implemented method of finding references to web pages posted on computer network the method using a database comprising sets of associated property values, the property values being any of classification values, contact values, geographic location values, hereinafter collectively referred to as CCG-data, and URL references, the method comprising the steps of:
- 35 a) receiving a query phrase including query relational expressions from a computer network,
 - b) parsing said query phrase and extracting each of said query relational expressions included therein,
 - c) from each extracted query relational expression, deriving a query field name,
 - 40 d) determining the type of said query relational expression by reference to its derived query field name,
 - e) relating said type of query relational expression so determined to one of the following query relational expression types: CCG-data type, other type,
 - f) provided said query relational expression is a CCG-data type, deriving a query relational operator and query value related to its query field name from said query relational expression,
 - 45 g) determining the type of said query value by reference to said query field name,
 - h) relating said type of query value so determined to a corresponding type of database property value,

- 5 i) locating database property values of said determined corresponding type which return a true value when tested against said query value using said query relational operator,
 j) extracting from said database a list of the URL references associated with the so located database property values.
11. A computer implemented method of finding sets of associated database property values the method using a database comprising sets of associated property values wherein each set includes at least two property values of different types, the property values being any of classification values, contact values, geographic values, hereinafter collectively referred to as CCG-data, the method comprising the steps of:
- 10 a) receiving a query phrase including query relational expressions from a computer network,
 b) parsing said query phrase and extracting each of said query relational expressions included therein,
 c) from each extracted query relational expression, deriving a query field name,
 d) determining the type of said query relational expression by reference to its derived query field name,
 e) relating said type of query relational expression so determined to one of the following query relational expression types: CCG-data type, other type,
 f) provided said query relational expression is a CCG-data type, deriving a query relational operator and query value related to its query field name from said query relational expression,
 g) determining the type of said query value by reference to said query field name,
 h) relating said type of query value so determined to a corresponding type of database property value,
 i) locating database property values of said determined corresponding type which return a true value when tested against said query value using said query relational operator,
 j) extracting from said database sets of associated database property values associated with the so located database property values.
- 20 12. A method of displaying a web page comprising at least one HTML encoded CCG phrase, the method comprising the steps of:
- 35 a) retrieving a web page from a computer network,
 b) parsing said retrieved web page to locate an HTML code indicative of the start of a CCG phrase,
 c) parsing said located CCG phrase and extracting successive CCG attributes contained therein until an HTML code indicative of the end of said CCG phrase is found,
 d) from each extracted attribute, deriving an attribute name,
 e) determining the type of said extracted attribute by reference to its derived attribute name,
 f) relating said type of attribute so determined to one of the following attribute types: database control, display control, CCG-data,
 g) provided said extracted attribute is not a database control type, deriving an attribute value related to its attribute name from said extracted attribute,
 h) determining the type of said attribute value by reference to said attribute name,
 i) relating said type of attribute value so determined to a corresponding type of parameter of a display-device-control-program.
- 40 45 50

- 5 j) writing said attribute value to said parameter, and
 k) where said type of attribute is a CCG-data type, causing said display-device-
 control-program to effect display of said attribute value on a display device,
 formatted and positioned according said display-device-control-program
 parameters whereby successive values of CCG-data of the CCG phrase are
 displayed.

ABSTRACT

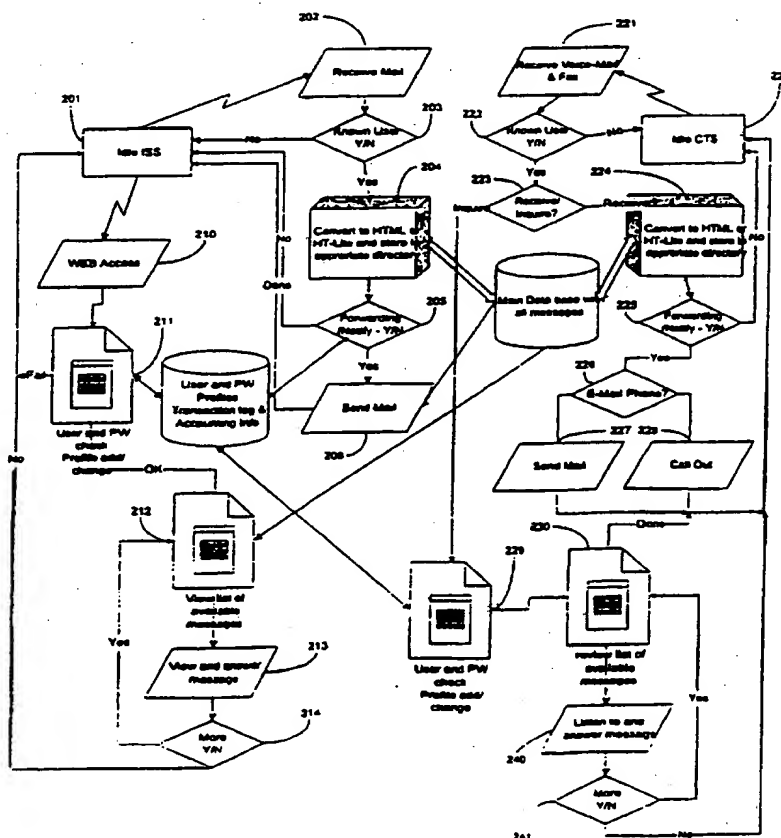
A system for automatically creating databases containing industry, service, product and subject classification data, contact data, geographic location data (CCG-data) and links to web pages from HTML, XML or SGML encoded web pages posted on computer networks such as the Internet or Intranets. The web pages containing HTML, XML or SGML encoded CCG-data, database update controls and web browser display controls are created and modified by using simple text editors, HTML, XML or SGML editors or purpose built editors. The CCG databases may be searched for references (URLs) to web pages by use of enquiries which reference one or more of the items of the CCG-data. Alternatively, enquiries referencing the CCG-data in the databases may supply contact data without web page references. Data duplication and coordination is reduced by including in the web page CCG-data display controls which are used by web browsers to format for display the same data that is used to automatically update the databases.

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

(51) International Patent Classification ⁶ : G06F 17/40		A1	(11) International Publication Number: WO 98/03928
			(43) International Publication Date: 29 January 1998 (29.01.98)
(21) International Application Number: PCT/US97/12628		(81) Designated States: CN, JP, European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).	
(22) International Filing Date: 18 July 1997 (18.07.97)		Published <i>With international search report.</i>	
(30) Priority Data: 08/685,025 23 July 1996 (23.07.96) US			
(71) Applicant: LEXTRON SYSTEMS, INC. [US/US]; 20264 Ljepava Drive, Saratoga, CA 94401 (US).			
(72) Inventor: KIKINIS, Dan; 20264 Ljepava Drive, Saratoga, CA 94401 (US).			
(74) Agent: BOYS, Donald, R.; P.O. Box 187, Aromas, CA 95004 (US).			

(57) Abstract

A web server system for delivering e-mail messages and other forms of digital documents converts incoming documents into Hypertext Markup Language (204) and stores them in an indexed database comprising directories and subdirectories. As requests are received (223) from users, HTML documents are retrieved from the directories and transmitted directly to the users with no need for conversion to another format. In a preferred embodiment directories are assigned to users and a user accesses a WEB page on the server to access digital documents. Attachments in this embodiment are related by hyperlinks.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece			TR	Turkey
BG	Bulgaria	HU	Hungary	ML	Mali	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MN	Mongolia	UA	Ukraine
BR	Brazil	IL	Israel	MR	Mauritania	UG	Uganda
BY	Belarus	IS	Iceland	MW	Malawi	US	United States of America
CA	Canada	IT	Italy	MX	Mexico	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NE	Niger	VN	Viet Nam
CG	Congo	KE	Kenya	NL	Netherlands	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NO	Norway	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	NZ	New Zealand		
CM	Cameroon	KR	Republic of Korea	PL	Poland		
CN	China			PT	Portugal		
CU	Cuba	KZ	Kazakstan	RO	Romania		
CZ	Czech Republic	LC	Saint Lucia	RU	Russian Federation		
DE	Germany	LJ	Liechtenstein	SD	Sudan		
DK	Denmark	LK	Sri Lanka	SE	Sweden		
EE	Estonia	LR	Liberia	SG	Singapore		

Integrated Services on IntraNet and Internet

Field of the Invention

5

The present invention is in the area of multimedia document handling and cross-media access of such documents based both on Internet, Intranet and Telephony networks.

10

Cross Reference to related Applications

This disclosure is related to patent application 08/629,475 by the same inventor.

15

Background of the Invention

Today many different electronic services are available for communication. Such services include, but are not necessarily limited to voice-mail, e-mail, paging, alpha paging, cellular phones, paging phones, fax machines and so forth. There are
20 also cross-linked services available, such as paging on digital cell phones, and the like. In general, however, each type of media is limited to one access, usually in very primitive manner.

Recently Motorola announced e-mail on cellular telephones: To use this service, a user calls a special number, and the saved e-mail is read over the phone to
25 the user. Such a service may be helpful in some cases, while not be very helpful in others. If, for example, a spreadsheet is attached, the spreadsheet cannot be read over the phone. Even if a spreadsheet could be converted, reading potentially hundreds of numbers over the phone will most likely result in several transcription errors, rendering the result basically useless.

30 What is needed are better devices and better methods, crossing traditional media boundaries.

One simple way to offer integrated services is to use a database on a server that is connected to the World Wide Web (WWW). Then, when data is requested, that data is called by invoking a so-called CGI-application (these are applications that are launched by a web page). The CGI application then sorts out data, and presents the
5 result as a dynamically-built web page. During a Comdex show on about November 14, 1995 Lotus, Inc. announced such a program for their Notes product. This addition allows users to read Notes. Others have followed since.

The problem remaining with such solutions is that most of them are proprietary and also slow, meaning that only a very limited number of users can be
10 serviced concurrently by one server. This is partly because a CGI application has to be launched for every request, invoking a database inquiry, which in all cases consumes substantial computer power and time.

15

Summary of the Invention

In a preferred embodiment of the present invention a web-server system for processing and providing digital documents, comprising: a receiver-converter for receiving digital documents and converting the digital documents to Hyper Text
20 Markup Language (HTML) format; a directory structure providing a database; and an index listing the contents of the database by directory structure. Upon receipt of a digital document, the receiver-converter converts the digital document into HTML format and stores it in the directory structure, and updates the index. In some embodiments the system further comprises an access program wherein database
25 directories are assigned to individual users and displayed as web pages. In this embodiment attachments to incoming e-mail are related to stored mail as hyperlinks to the web page.

A method is provided comprising steps of (a) making a database on a server composed of directories assigned to users; (b) receiving digital documents at the
30 server; (c) converting the digital documents to Hypertext Markup Language (HTML) format; and (d) storing the HTML digital documents in the directories. In some embodiments the method further comprises steps for: (e) receiving a request from a

user; (f) retrieving a document from the database in HTML format; and (g) transmitting the document to the user over the Internet.

In various embodiments, assuming servers of relatively equal computing power, by using a directory structure instead of an integrated database, and pre-
5 converting documents to HTML format prior to storing for later retrieval by a user, more users can be served at a faster pace than can be served in conventional systems.

Brief Description of the Drawing Figures

10

Fig. 1 is a generalized topology example showing arrangement and connectivity of equipment in an embodiment of the present invention.

Fig. 2 is a flowchart illustrating processes and operations in practicing embodiments of the present invention.

15

Description of the Preferred Embodiments

The present invention in various embodiments differs from the prior art in that
20 a database is not used, as described above in the Background section. When a database is used with the Internet (WWW), once data is extracted, it must be converted to HTML (Hyper Text Markup Language) before the data can be transmitted on the Internet. This is typically done as a function of the CGI application called. Instead, in embodiments of the present invention, the digital documents
25 (mails) are as HTML files in a directory structure representing the database. In addition, in some embodiments, even the index is kept in a HTML file, and the index is continuously updated as messages come in.

In an alternative embodiment small downloadable modules, in technologies such as JAVA or similar, are provided on a server connected to the WWW. A user
30 first downloads the HTML index and a small application to handle it, then executes actual index searches on the user machine. Once a file or files are located in the index a request is set over the Internet to access the file or files from the server

In one embodiment the existing "Send Mail of the UNIX operating system on a server is modified in a way that incoming mail is converted into HTML files, and then stored in appropriate pre-arranged directories. An index file is then updated. If the incoming mail has attachments, they are stored in the same directory and can have a hyperlink from the mail page. A user may then either view or download the attachment(s).

Additional functions, such as address book, sending mail etc. are provided in embodiments of the invention by using a Java applet having a relatively small user interface. The applet can directly access files containing addresses and insert them into messages and so forth.

In some embodiments of the invention, to facilitate adding of addresses, the addresses can be marked as well on the message, and by clicking on the addresses a user can cause the addresses to be transferred into the list. The address list also contains, in some embodiments, phone numbers and addresses (snail mail), that can be launched into other applications such as a phone dialer. This feature is very attractive in conjunction with such things as voice-mail, video-mail etc. Players and auxiliary tools may be launched to connect a user with a calling party, or to allow a user to leave voice-mail and or video-mail messages.

On the receiving end, where a user is using a system according to an embodiment of the present invention, voice-mail and video-mail are converted when received into one or several 'standard' formats, so that when the user wants to view it, no long delays are incurred. Without this feature a user may have to launch a CGI-controlled search through a database, followed by on-the-fly conversion, which can consume a substantial amount of CPU power.

In embodiments of the present invention all files are prepared when arriving, such that the user when checking, can just browse. By using an HTTPS server, security is provided by standards already established on the Internet. This feature allows more users on a single server, which ultimately reduces costs dramatically.

In an ideal setup, the user can go to a web-page, and open his own account all by himself, since only name, password and credit card (or some other form of payment) are needed. There are no IP addresses etc. to worry about. Additionally, The user may also open up his own web page much like the same web-page referred to

above, and then upload through a secure HTTPS transaction new pages that he created on his own system.

It will be apparent to those with skill in the art that there are many alterations that might be made in the embodiments described without departing from the spirit
5 and scope of the invention. For example, there are many ways directory structures may be provided and many ways individual programmers might furnish code to accomplish the modules of the invention. There are similarly many sorts of platforms and data links that may be used in practicing embodiments of the invention. The invention is limited in scope only by the claims which follow.

What is claimed is:

1. A web-server system for processing and providing digital documents, comprising:
a receiver-converter for receiving digital documents and converting the digital
5 documents to Hyper Text Markup Language (HTML) format;
a directory structure providing a database; and
an index listing the contents of the database by directory structure;
wherein, upon receipt of a digital document, the receiver-converter converts
the digital document into HTML format and stores it in the directory structure, and
10 updates the index.
2. A web-server system as in claim 1 further comprising an access program wherein
database directories are assigned to individual users and displayed as web pages.
- 15 3. A web-server system as in claim 1 wherein attachments in incoming e-mail are
related to stored mail as hyperlinks to the web page.
4. A method for providing integrated digital document services to users, comprising
steps of:
20 (a) making a database on a server composed of directories assigned to users;
(b) receiving digital documents at the server;
(c) converting the digital documents to Hypertext Markup Language (HTML)
format; and
(d) storing the HTML digital documents in the directories.
- 25 5. The method of claim 4 further comprising steps for:
(e) receiving a request from a user;
(f) retrieving a document from the database in HTML format; ad
(g) transmitting the document to the user over the Internet.

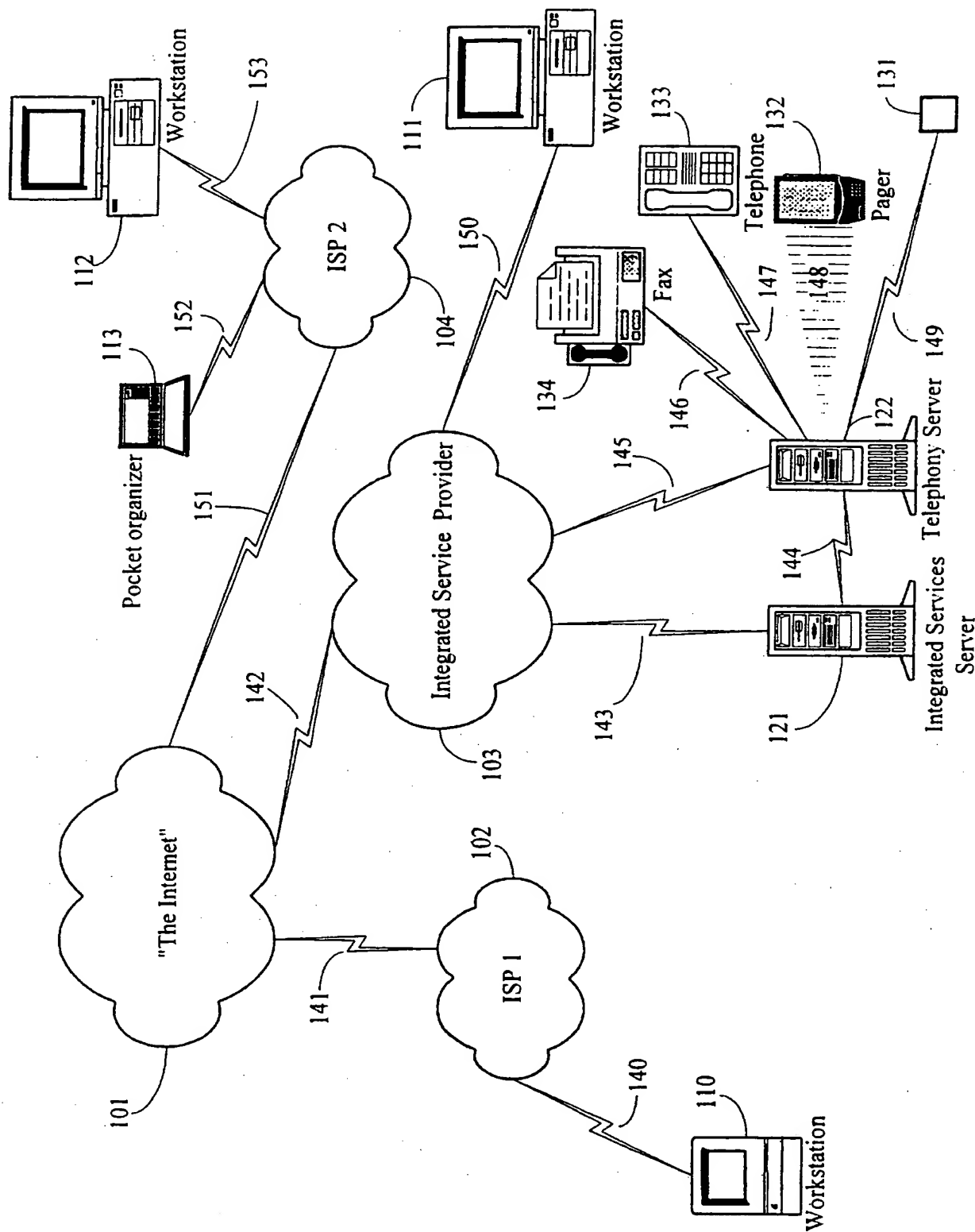


Fig. 1 - Topology Example

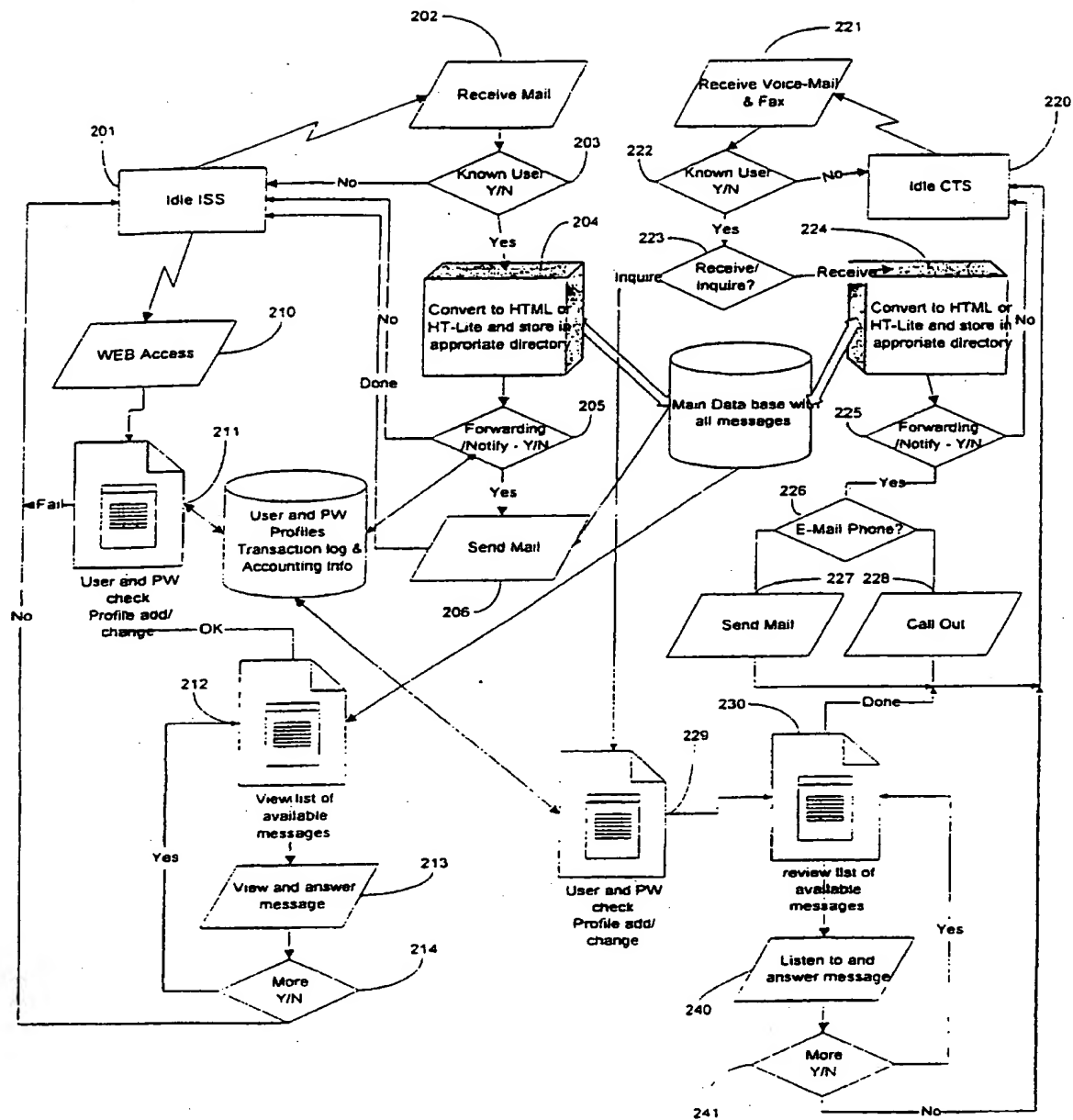


FIG. 2

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US97/12628

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) : G06F 17/40

US CL : 395/774

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 395/774, 762, 610; 358/402, 403

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

IEEE CD-ROM, Computer Select 1995-1996 CD-ROM, APS

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X,P	US 5,649,186 A (FERGUSON) 15 JULY 1997 (15.7.97) SEE ABSTRACT	1-5
Y,E	US 5,654,886 A (ZERESKI ET AL.) 05 AUGUST 1997 (5.8.97) SEE ABSTRACT	1-5
Y,P	US 5,627,997 A (PEARSON ET AL.) 06 MAY 1997 (6.5.97) SEE ABSTRACT	1-5
Y,P	US 5,623,589 A (NEEDHAM ET AL.) 22 APRIL 1997 (22.4.97) SEE ABSTRACT	1-5
Y,P	US 5,608,874 A (OGAWA ET AL.) 04 MARCH 1997 (4.3.97) SEE ABSTRACT	1-5
Y,P	US 5,608,446 A (CARR ET AL.) 04 MARCH 1997 (4.3.97) SEE ABSTRACT	1-5

☒ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

* ..	Special categories of cited documents:	* ..	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
* A *	document defining the general state of the art which is not considered to be part of particular relevance	* X *	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
* E *	earlier document published on or after the international filing date	* Y *	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
* L *	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	* & *	document member of the same patent family
* O *	document referring to an oral disclosure, use, exhibition or other means		
* P *	document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search

26 AUGUST 1997

Date of mailing of the international search report

09 OCT 1997

Name and mailing address of the ISA/US
Commissioner of Patents and Trademarks
Box PCT
Washington, D.C. 20231

Facsimile No. (703) 305-3230

Authorized officer

ANTON W. FETTING

Telephone No. (703) 305-8449

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US97/12628

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y,P	US 5,577,108 A (MANKOVITZ) 19 NOVEMBER 1996 (19.11.96) SEE ABSTRACT	1-5
Y,P	US 5,553,281 A (BROWN ET AL.) 03 SEPTEMBER 1996 (3.9.96) SEE ABSTRACT	1-5
Y	US 5,530,852 A (MESKE JR. ET AL.) 25 JUNE 1996 (25.6.96) SEE ABSTRACT	1-5
Y	US 5,406,557 A (BAUDOIN) 11 APRIL 1995 (11.4.95) SEE ABSTRACT	1-5
Y	KIKUCHI ET AL., USER INTERFACE FOR A DIGITAL LIBRARY TO SUPPORT CONSTRUCTION OF A VIRTUAL PERSONAL LIBRARY, PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON MULTIMEDIA COMPUTING AND SYSTEMS, 17 JUNE 1996, P. 429-432 , SEE P.429	1-5
Y	WILKINSON, HARMONIC CONVERGENCE, PC WEEK, 11 MARCH 1996, V.13,N.10,P.15-16, SEE P. 15	1-5
Y	CROTTY, NETSCAPE NAVIGATOR SHATTERS STATIC WEB PAGES, MACWORLD, DECEMBER 1995, V.12, N.12, P.34-35, SEE P. 34	1-5
Y	SEMILOF, PROTOTYPE E-MAIL WARE DRAWS MIXED REVIEWS, COMMUNICATIONSWEEK, 17 APRIL 1995, N.553, P. 15, SEE P. 15	1-5

XP-002193377

P.D. 12-1997	7
p. 32-38 =	

Extracting Entity Profiles from Semistructured Information Spaces

Robert A. Nado Scott B. Huffman

Price Waterhouse Technology Centre
68 Willow Road
Menlo Park, CA 94025-3669
{nado, huffman}@tc.pw.com

Abstract

A semistructured information space consists of multiple collections of textual documents containing fielded or tagged sections. The space can be highly heterogeneous, because each collection has its own schema, and there are no enforced keys or formats for data items across collections. Thus, structured methods like SQL cannot be easily employed, and users often must make do with only full-text search. In this paper, we describe an approach that provides structured querying for particular types of *entities*, such as companies and people. Entity-based retrieval is enabled by *normalizing* entity references in a heuristic, type-dependent manner. The approach can be used to retrieve documents and can also be used to construct entity profiles – summaries of commonly sought information about an entity based on the documents' content. The approach requires only a modest amount of meta-information about the source collections, much of which is derived automatically.

1 Introduction

Decentralized information sharing architectures like the World Wide Web and Lotus Notes make it easy for individuals to add information, but as the space grows, retrieval becomes more and more difficult. *Semistructured* information sharing systems, including Lotus Notes™ and a variety of meta-tagging schemes being developed for the World Wide Web (e.g. Apple's Meta Content Framework [Guh97]), address part of this problem by providing the ability to structure local parts of the information space. In a semistructured information space, documents are sectioned into weakly-typed fields according to user specifications, and documents with the same field

structure can be grouped into collections. Within a collection, field values can be used as indexes for easier retrieval.

Unfortunately, semistructuring document collections does not solve the problem of retrieving information across a large information space. Even if individual collections are well-designed for retrieval, users can be overloaded with the sheer number of collections. Retrieval across the entire space is difficult because it is highly heterogeneous. Each collection has its own local schema, and there are no enforced keys or formats for data items within or across collections.

Our work addresses the problem of finding and integrating useful information across collections in large semistructured information spaces. Our goal is to provide querying that is more powerful and precise than full-text search, but without requiring the collections to be strongly typed, data normalized, and fully mapped to a global schema, as methods like multidatabase SQL require. In this paper, we focus on the retrieval of integrated summaries of useful information (entity profiles), drawn from multiple, heterogeneous document collections.

Our approach is to provide high quality retrieval of information related to important *entities* in the information space. In our organization (a large professional services firm), important types of entities include people, companies, and consulting skills. A review of our largest collections revealed that nearly always, references to important entities are fielded rather than buried in free-running text. Because the same entity can be referred to in many different ways across a heterogeneous information space, our entity retrieval system *normalizes* references to entities in a heuristic, type-dependent manner. For instance, the person names "Mr. Bob Smith", "Smith, Robert", and "R. J. Smith" are normalized such that a query for any

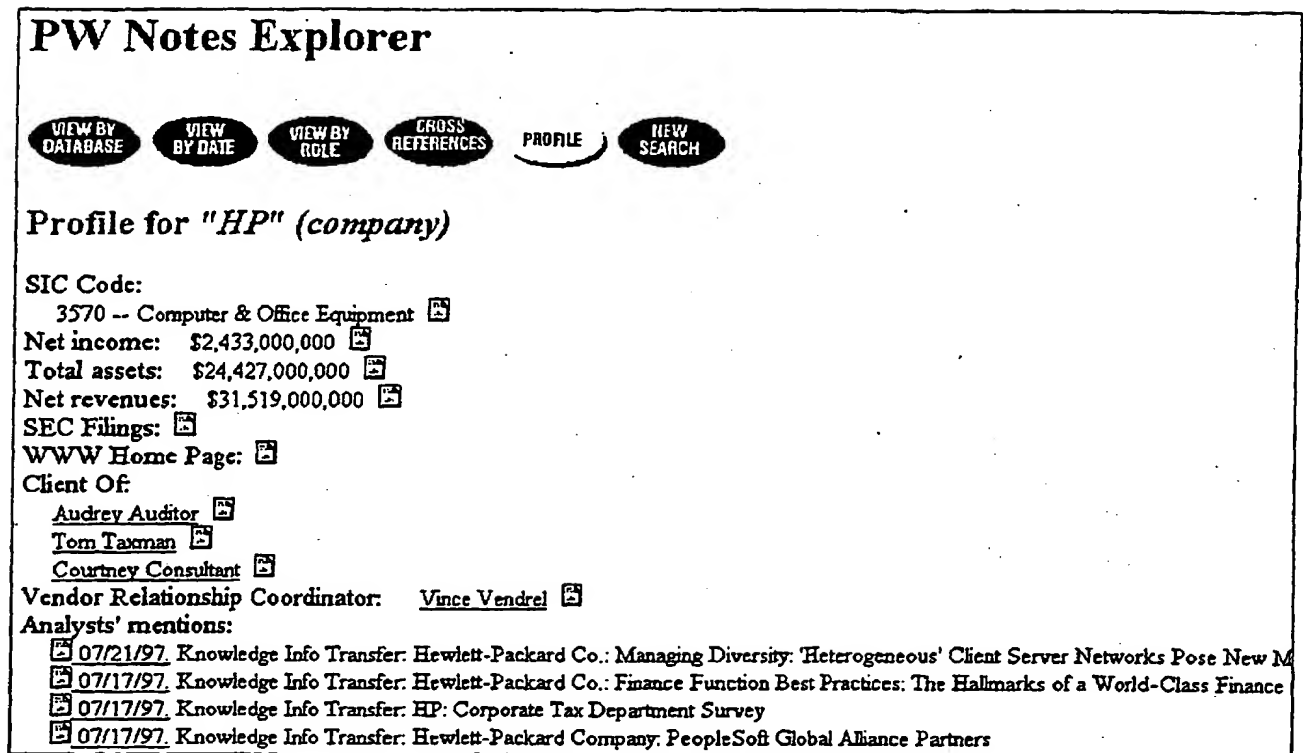


Figure 1: Results of NX Profile Search

one (or a number of other possible forms) will retrieve documents containing any of them.

We have implemented an entity-based retrieval system called *NX* (for Notes Explorer) that operates over a large semistructured information space. The space currently includes over one hundred corporate Lotus Notes collections and a small set of web collections, together containing about 300,000 documents. *NX* provides full-text search, entity-based document retrieval for people, companies, and skills, and profile extraction for people and companies. It is delivered over an intranet using HTML.

A key hypothesis behind this work is that a *relatively small amount of meta-information* – much less than that required to normalize and map collections to a comprehensive global schema – *can give a large gain in query power and precision* over knowledge-free methods like full-text search. *NX* is one illustration of this hypothesis. It requires only a modest amount of meta-information about each collection – an indication of fields containing entities in various semantic categories and pairs of fields that stand in specific semantic relations – and uses it to produce a dramatic improvement in retrieval quality for entity-related queries. Much of the required meta-information can actually be inferred automatically

based on field names and data within the collections, using a simple heuristic classifier.

In what follows, we first motivate the task of generating entity profiles with a real-world example. Next, we describe the main components of our retrieval system. We conclude by discussing related and future work.

2 Entity-based retrieval

In a corporate setting, information in different documents is frequently linked through references to entities with business importance, such as people and companies. Often, users search for information about *particular* entities (e.g., "What is Bob Smith's phone number?" or "Who's the manager for the XYZ Co. account?") as opposed to ungrounded, aggregate queries across sets of entities (e.g. "Show me all managers with more than five clients over \$5 million in sales"). We designed *NX* to support this type of search.

Consider a typical example from our organization. A staff member is writing a proposal to XYZ Company for some consulting work. She needs answers to questions like:

- (a) How large is XYZ Company? E.g., what are their assets, revenues, etc.?

- (b) Does our organization have a prior relationship with XYZ? Have we done other consulting work for them in the past?
- (c) If so, who did that work, and how can they be contacted?

Each question refers to entities of various types – XYZ Company, staff members, phone numbers, etc. – and these entities may be referred to differently in different documents. Some questions involve information that may be found in many collections of the same type – e.g., information about prior work for XYZ (b) might be found in numerous collections containing client engagements. Others involve linking information about XYZ with information about another entity -- e.g., question (c) requires finding staff names in documents that list XYZ engagements, and then finding contact information for those staff names.

Figure 1 displays the results of a profile search in NX given "HP" as a company name search string.¹ Normalization allows NX to retrieve information from documents that mention "Hewlett Packard", "Hewlett-Packard, Inc.", etc., as well as "HP". The headings (e.g., "SIC Code:" and "Client Of:") list specific values that have the specified relationship to the company of interest. These values are drawn from multiple documents in different collections; the square document icons are hyperlinks to the source documents. In the case of values representing people and companies, the value (e.g., "Audrey Auditor") is also displayed with a hyperlink that initiates a profile search on that value. This allows, for example, contact information to be found for people who have "HP" as a client. Other headings (e.g., "SEC Filings:" and "Analysts' mentions:") are followed only by document links, as it is the document as a whole that is of interest -- not specific information extracted from it.

3 Extracting Entity Profiles in NX

This section describes the major components of NX that are used to support its profile search capability:

- Semi-automatic field classification.
- Entity normalization.
- Definition of a partial global schema
- Extraction of profile information from entity indexes
- Detection and resolution of profile ambiguity

¹ Actual people names have been replaced in the HTML generated by Notes Explorer to preserve privacy.

3.1 Semi-automatic field classification

To build an index of entity references of different types, we must identify where those types occur within collections. NX's field classifier uses field names and sample values from a collection to classify fields as containing entity types (people's names, company names, phone numbers, dollar amounts, etc.) and identifiable semantic *roles* that they play within the collection, e.g., partner on an engagement, client company, or vendor company. The current version recognizes person names, company names, telephone numbers, geographic locations, office names, and dollar amounts. As classification is not 100% accurate or complete, a Web browser interface is provided to alter the entity and role types for each collection's fields.

3.2 Entity Normalization

In a standard relational database, tuples from different tables that contain information about the same entity each contain a *key* for that entity allowing the tables to be joined. In a semistructured document space, however, there are rarely unique keys shared by collections. Rather, entities are referred to within text strings in a variety of formats, with a variety of synonyms and abbreviations.

Therefore, to allow search over entities, entity references must be normalized and matched (as in [HS95]). For maximum retrieval speed, NX normalizes entity references at indexing time. The normalization is heuristic, using formatting knowledge and synonym tables specific to each entity type. NX's entity index stores both the original form and a normalized form of each entity reference. At retrieval time, a normalized form of the user's search string is created and used to retrieve matches from the normalized entity index. In some cases, values are only partially normalized, and the original forms of retrieved matches and the search string are compared to verify the match.

In addition, pre-processing is required to find the portions of the input string containing entity references. Often, a field will contain multiple entity values in a single string, with spurious information interspersed. For example, a typical person name field value might be "Bob J. Smith Jr. – managing partner; Sue Jones, 415-555-1212, Palo Alto." NX's normalization routines extract "Bob J. Smith Jr." and "Sue Jones" out of this field value.

NX's field classification and normalization routines are described in more detail in [HB97].

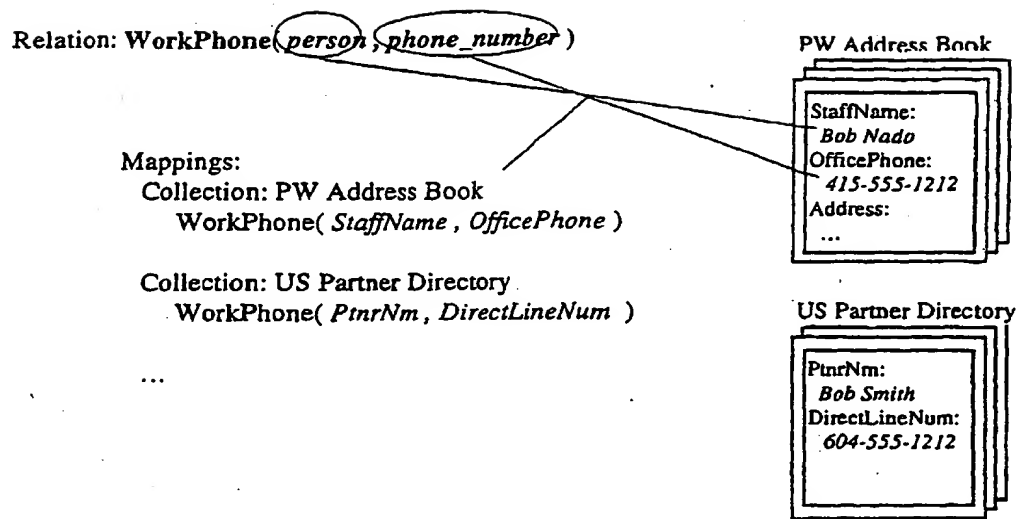


Figure 2: Mapping a Predicate to Collection Fields

3.3 Definition of a Partial Global Schema

The profile search capability of NX is based on a global vocabulary for describing the types of information that may be found about an entity in the different information sources that are available. No attempt is made to define a complete global schema characterizing all of the relations that might be extracted from individual collections. Rather, the global schema used by NX is partial – containing only enough meta-information to support the desired entity profiles. Currently, the global vocabulary includes sorted, binary predicates of two types. An *entity predicate* represents a relationship between two entities. For example, "Work Phone" is an entity predicate representing the relationship between a person and a phone number where that person may be reached at work. Sorts (entity types) are assigned to the domain and range arguments of the "Work Phone" predicate -- "Person" and "Phone Number" -- restricting the applicability of the predicate. The other type of predicate, called a *document predicate*, represents a relationship between an entity and a document that is "about" that entity. For example, "Resume" is a document predicate relating a "Person" and a resume document.

In addition to declaring domain and range sorts, each predicate must be mapped to the relevant fields in collections that locally instantiate the predicate. For example, in the *PW Address Book* collection, the "Work Phone" predicate is mapped to a domain field called "StaffName" and a range field called "OfficePhone". Other collections may also have

information relevant to the "Work Phone" predicate but use different fields to record the person name and the phone number (see Figure 2). An entity predicate may be mapped to multiple pairs of domain and range fields in a single collection. Document predicates have a simpler mapping, requiring only a domain field in each relevant collection.

Currently, the mapping of predicates to fields in collections is performed manually using a Web browser interface. The interface narrows the set of candidate collections and fields for each predicate by exploiting the entity types assigned to fields by NX's field classifier. A collection can be ignored when mapping a predicate if it does not contain fields with entity types matching both the domain and range sorts of the predicate. Given an eligible collection, candidates for the domain and range fields are narrowed to those whose entity types match the domain and range sorts of the predicate. The interface allows the predicate mapping process to be performed in a small amount of time, typically less than a half hour per collection.

3.4 Extraction of profile information from entity indexes

A profile for a particular category of entity is defined by listing the global predicates that should make up the profile in the order in which they should be displayed in the results page of a profile search. Information can be associated with individual predicates through a Web browser interface to control the formatting, number, and sorting of profile results displayed for the predicates.

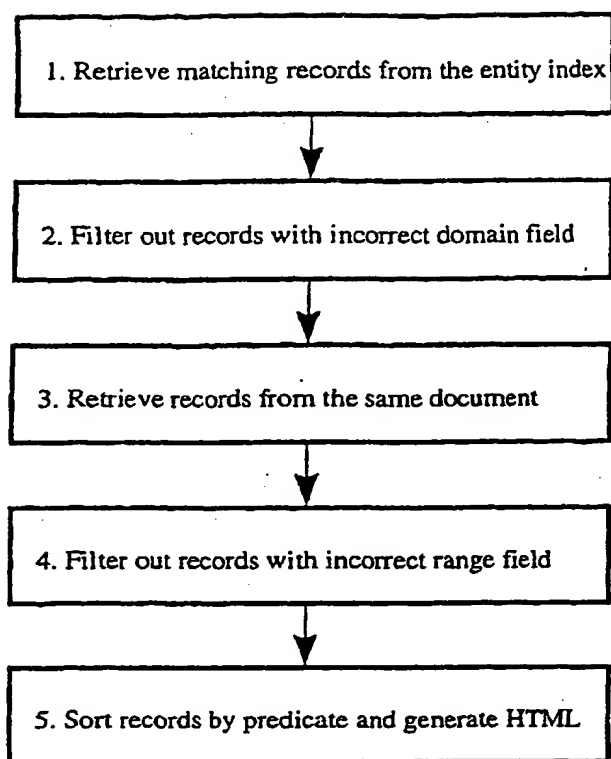


Figure 3: Profile Information Extraction

Retrieving an entity profile involves five steps as depicted in Figure 3. First, NX retrieve records from the entity index whose normalized field values match the normalized forms of the search string and that have the correct entity type. Next, NX filters out records retrieved in Step 1 whose field is not a domain field for any predicate in the profile. For each record *A* resulting from step 2, NX retrieves records from the entity index that originate in the same document. In step 4, NX filters out records retrieved for *A* in Step 3 whose field is not a range field corresponding to *A*'s field as a domain field for one of the profile predicates. Finally, NX sorts the remaining records by profile predicate and generates an HTML page displaying the results for each predicate.

Because they have been normalized, the results found for a particular profile predicate can be properly grouped independently of how they were referred to in the source documents. In essence, this uses the normalized entity index as a simple data warehouse, enabling an aggregation over entities in document sets.

3.5 Profile ambiguity

Given a particular search string, NX may find references in documents to more than one distinct

entity, each of whose names match the search string. For example, when "Bob Smith" is supplied as the search string, matches may be found that give information about both Robert A. Smith and Robert S. Smith. This problem of ambiguous profile searches can only be addressed heuristically, as entities do not have unique keys across collections.

In some cases, however, reference lists of entities can be used to aid in disambiguation. For person names within our firm, for instance, there are "address books" mapping each person's name to a unique email address. Generally, a PW staff member will have exactly one entry in one of the address books that exist for each PW firm around the world. If more than one match is found for the search string in the collection of address books, the user is asked to select one of the address book entries in order to refine the search. This is illustrated in Figure 4 for a profile search on "Bob Smith".

The selected address book entry often gives a more specific search string with which to continue the search. In addition, the address book entry may give other information about the chosen person (such as work office) that may be used to filter out documents that contain conflicting information.

4 Discussion and Future Work

As described in [HB97], we have evaluated NX's entity-based retrieval through a comparison to standard full-text search, finding that it produces much more precise result sets than full-text search for important classes of queries. To date, we have not explicitly evaluated the entity profiling capability. It may be difficult to use traditional IR evaluation metrics like precision and recall over such a large and diverse information space. Rather, we plan to evaluate profiles' usefulness to end users, through user feedback and surveys.

The goal of our work is to provide better information retrieval across a large semistructured space than full-text search, while avoiding excessive meta-information overhead. Our approach is based on observing that in an information space used by a particular organization, important entity types link information together and can be used as a central retrieval cue. This data-driven approach can be contrasted with schema-driven approaches used by multidatabase systems (e.g., [ACHK93]), and similar systems attempting to integrate structured world-wide web sources [LRO96, FDFP95]. In schema-driven approaches, each local schema is mapped to a central global schema, and mapping rules are used to translate between data formats used by different

PW Notes Explorer



Choose a person to profile:




-  Robert Smith: PW Hobart, Robert Smith (PW Australia Address Book) Select
-  Rob Smith: Rob Smith; UK, Southampton (PW Europe Address Book) Select
-  Robert S. Smith: Austin, Texas; Robert S. Smith (PW Name & Address Book) Select

Figure 4: Ambiguous Profile Search

sources (e.g. [CHS91]). These approaches are appropriate for relatively small numbers of tables where the data within each table is well-specified; however, semistructured information spaces can include hundreds of sources, and data even within single sources can have multiple formats. A schema integration phase would be burdensome in such a large space [GMS94]. Instead, NX relies on heuristics to categorize fields into a small number of entity and role types, and normalizes entity values for retrieval. The resulting retrieval system makes it practical to encompass a greater number and variety of data sources than multidatabase systems, although the query language is less general because queries must refer to a specific entity.

Topics to be addressed by future work include:

- extending profiles to other entity types such as service lines and skills,
- customizing profiles to meet the requirements of particular classes of users,
- using information about recency and reliability to resolve conflicts in information retrieved as part of a profile, e.g., multiple office phone numbers retrieved for a person
- performing inference in the determination of profile results that combines information from several documents, e.g., determining a person's office telephone number from his assigned office and that office's main switchboard number
- developing automated techniques for mapping global schema predicates to pairs of collection fields by exploiting abstract classifications of collections, e.g., "directory" collections are more likely to contain a person's phone number; client engagement archives

are more likely to contain the names of a person's clients.

- extending the information available as part of a profile by developing additional extraction and summarization methods, e.g., producing a summary of a person's key skills from resume documents

6 Conclusion

Semistructured systems are an intermediate point between unstructured collections of textual documents (e.g., untagged Web pages) and fully structured tuples of typed data (e.g., relational databases). Based on observing how information is typically retrieved and used within our organization, we have developed an entity-based retrieval system over a large semistructured information space. The system incorporates semi-automatic classification of fields, normalization of field values, and structured retrieval of commonly required information in the form of entity profiles. For typical queries containing entities, the system provides much more focused and normalized retrieval than full-text search.

References

- [ACHK93] Y. Arens, C.Y. Chee, C.N. Hsu, and C.A. Knoblock. Retrieving and integrating data from multiple information sources. *Intl Journal on Intelligent and Cooperative Information Systems*, 2(2):127-158, 1993.
- [CHS91] C. Collet, M.N. Huhns, and W. Shen. Resource integration using a large knowledge base in Carnot. *IEEE Computer*, pp. 55-62, December 1991.

[FDFP95] A. Farquhar, A. Dappert, R. Fikes, and W. Pratt. Integrating information sources using context logic. In *Working Notes of the AAAI Spring Symposium on Information Gathering from Heterogeneous Distributed Environments*. AAAI, 1995.

[GMS94] C.H. Goh, S.E. Madnick, and M.D. Siegel. Context Interchange: Overcoming the challenges of large-scale interoperable database systems. In *Proceedings of the 3rd International Conference on Information and Knowledge Management*. 1994.

[Guh97] R.V. Guha. Meta Content Framework: A Whitepaper (Draft). Apple Computer. Available at URL <http://mcf.research.apple.com/wp.html>, 1997.

[HB97] S. Huffman and C. Baudin. Toward Structured Retrieval in Semi-structured Information Spaces. In *Proceedings of the 1997 International Joint Conference on Artificial Intelligence*, 1997.

[HS95] S. Huffman and D. Steier. Heuristic joins to integrate structured heterogeneous data. In *Working notes of the AAAI Spring Symposium on Information Gathering in Heterogeneous Distributed Environments*. AAAI, 1994.

[LRO96] A. Levy, A. Rajaraman, and J. Ordille. Query-answering algorithms for information agents. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pp. 40-47, 1996.

XP002949327

D. 1998
1-25 25

NoDoSE - A tool for Semi-Automatically Extracting Structured and Semistructured Data from Text Documents.

Brad Adelberg*

Abstract

Often interesting structured or semistructured data is not in database systems but in HTML pages, text files, or on paper. The data in these formats is not usable by standard query processing engines and hence users need a way of extracting data from these sources into a DBMS or of writing wrappers around the sources. This paper describes NoDoSE, the Northwestern Document Structure Extractor, which is an interactive tool for semi-automatically determining the structure of such documents and then extracting their data. Using a GUI, the user hierarchically decomposes the file, outlining its interesting regions and then describing their semantics. This task is expedited by a mining component that attempts to infer the grammar of the file from the information the user has input so far. Once the format of a document has been determined, its data can be extracted into a number of useful forms. This paper describes both the NoDoSE architecture, which can be used as a test bed for structure mining algorithms in general, and the mining algorithms that have been developed by the author. The prototype, which is written in Java, is described and experiences parsing a variety of documents are reported.

Keywords: data extraction, semistructured data, structure mining, wrapper induction.

1 Introduction

The amount of useful semistructured data [Abi97] on the web continues to grow at a torrid pace. Users would like to gain conventional database system functionality on this data such as sophisticated querying and reporting. This has spurred a recent flurry of work [KWD97, AK97a, HGMC⁺97, DEW97] on generating wrappers around such sources, either manually or with software assistance, to bring the new data within the reach of general query tools. It is important to note, however, that a vast quantity of semistructured data stored in electronic form is not in highly formatted HTML pages but in text files on local file systems. Examples are mail files, code and code documentation, configuration files, logs of program activity, phone lists, etc. Further, there is a huge collection of semistructured information in print, that when scanned and OCR'd will be in a plain text file, not one with HTML tags. Thus if we want to extract all of the data that's important to users through a query interface, we need to focus on something more general than HTML files — plain text files. Since HTML files are a special case of text files, a tool that handles text files will also handle HTML files.

Extracting information from text files is harder than for HTML files for three reasons:

1. Since text files do not generally contain markup tags, there are usually fewer structural clues and those that are present are not known a priori.

*Northwestern University Computer Science Department. Email: adelberg@cs.nwu.edu. Address: 1890 Maple Avenue, Evanston IL 60201. Telephone: (847) 467-2129. Fax: (847) 491-5228.

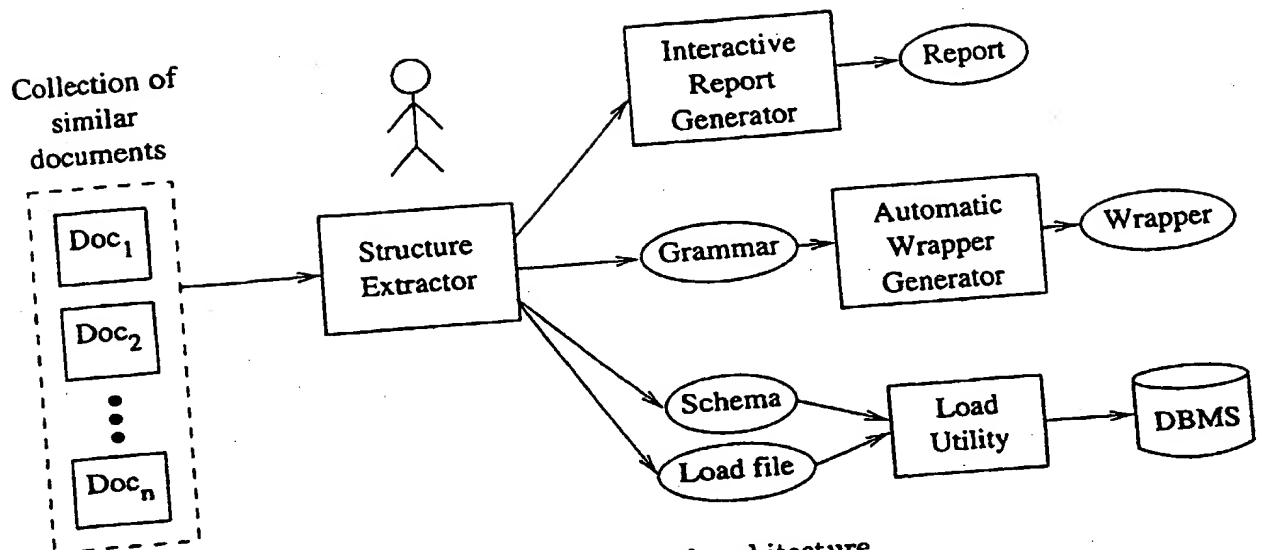


Figure 1: User level architecture.

2. Not all of the structural elements will be separated by markers (such as tags). Thus an extraction tool may have to consider parsing rules such as "list elements are exactly three lines long" as opposed to HTML where they begin after `` and end before another list tag or an end list tag (i.e. ``).
3. Now that most HTML pages are composed using authoring tools, their format (e.g. spacing, capitalization, etc.) is very regular. Many text files, in contrast, are typed by humans and are thus less regular and may contain errors. Also, text files scanned from documents will almost undoubtedly contain misrecognized characters.

Given the complexity of parsing general text files, the likelihood of a fully automatic extraction system like those proposed for HTML files [KWD97] seems remote. With no clues about the format of a document, the system will not be able to differentiate useful data from junk. We note, however, that a few clues can go a long way: If a user indicates a few of the regions of a document that are interesting it may be possible to identify similar regions automatically.

Thus the approach we have taken is to build a *semi-automatic* system, one that cooperates with the user to extract the data. Figure 1 illustrates how the process would work. The input to the extractor is a set of text files (documents) that are instances of the same document type. For example, the files might be reports generated by a weekly file backup program for the past year. Using a GUI, the user hierarchically decomposes the files, outlining their interesting regions and then describing their semantics. This task is expedited by a mining component that attempts to infer the grammar of the file using the information the user has input so far. A detailed example of this process is presented in Section 2.

Once the format of the document type has been determined and verified by successfully parsing all of the input documents, the extractor should be able to generate different types of output:

1. Using a simple interactive report generator, the user could filter the data from the input documents and write it out in a new format. For example, the user may want to convert the

input into a comma delimited file suitable for importation into a data analysis program or spreadsheet.

2. If the extracted data is to be stored in a database, a schema file (if the data is structured) and a load file suitable for the load utility that comes with the DBMS can be generated.
3. If the input documents are to be exposed through a query interface, the Lex and Yacc code needed by a wrapper can be generated.

In this paper we describe the design and implementation of this extraction tool — NoDoSE, the Northwestern Document Structure Extractor. We cover both the NoDoSE architecture (in Section 3), which can be used as a test bed for structure mining algorithms in general, and the algorithms for inferring parsing rules that we have developed (in Section 4). A first version of NoDoSE has been implemented and is described in Section 5 along with a description of our experiences using it to extract data from a variety of documents.

2 Example

To illustrate the use of NoDoSE we consider the problem of analyzing the results of simulation experiments. Although recent work on data extraction [KWD97, AK97a, HGMC⁺97] has focused on web pages, we choose simulation output here since it is meant to be human readable and not program readable. Thus it has fewer structural clues (such as tags) and is more difficult to determine parsing rules for. Also, we anticipate that a system capable of inferring parsing rules for plain text documents will be able to handle HTML documents as well.

An example of human readable simulator output (which was generated by DeNet [Liv90]) is shown in Figure 2¹. Most tools for storing and analyzing data, such as database systems, spreadsheets, and plotting programs, cannot handle files this complicated. Instead, the output file must be converted into a more regular file (i.e. tabular) before it can be processed. Typically, this conversion is performed by a hand-coded program in awk, sed, perl, or some other scripting language. Using NoDoSE, the conversion can be performed quickly and without any coding expertise.

There are three steps to the conversion process:

1. Decide on how to model the data in the documents.
2. Hierarchically decompose the files, mapping regions of the files into components of the chosen model.
3. Specify how the extracted data is to be output.

We describe each of the three steps in the sections below.

¹The output as shown is slightly modified from the original. For illustration purposes, node results relating to confidence intervals were removed since they cannot be represented as a <average,stdev,num> triple. NoDoSE can parse the original files if a different data model is chosen.

```

DeNet(V1.6) simulation started on Mon Sep 5 14:41:31 1994
...
Attributes of node # 0 (Principal)
    simTime - 5.000000E+02
Attributes of node # 1 (usource)
    arrivalRate - 2.000000E+02
    meanSkewL - 1.000000E-01
    meanSkewH - 1.000000E-01
...
Attributes of node # 5 (tsink)
    batchSize - 100
    confidenceLevel - 9.500000E-01
    confidenceInterval - 1.000000E+00
Sample Results Node # 1 (usource)
    1 numUpdates - (avg) 1.000000E+00 - (std) 0.000000E+00 - (num) 100431
Sample Results Node # 2 (tsource)
    2 numJobs - (avg) 1.000000E+00 - (std) 0.000000E+00 - (num) 2455
...
Sample Results Node # 5 (tsink)
    5 misDL - (avg) 9.295315E-01 - (std) 2.559869E-01 - (num) 2455
    5 missedStale - (avg) 9.295315E-01 - (std) 2.559869E-01 - (num) 2455
DeNet(V1.6) simulation terminated on Mon Sep 5 14:42:24 1994
Total CPU usage      0.896 Minutes. (user      0.894 ; system      0.002 )

```

Figure 2: Simulation output example.

2.1 Modeling the documents

Before extracting data from the documents the user must decide how to model the data. One possibility for the data, and the one that will be used in this example, is shown in Figure 3. Documents are of type `SimulationRun` and contain three top-level components: a timestamp, a list of input parameters for each simulation node, and a list of measured results for each node. The parameters for each node (part of `NodeParams`) are represented as a list of `<name,value>` pairs. This has the benefit of a very regular structure but the disadvantage that the type information about parameters is being lost since every value is modeled as a string. The structure is so regular, in fact, that the output of any simulator written in DeNet can be modeled using this schema.

Instead of representing all of the different simulator nodes in a generic way, we could also create a new class for each. This solution has the benefit that we can model the particular parameters of each node and their real types. It also has two problems: any change to the simulator will require a change to the model, and the grammar derived for the output of one simulator cannot be used on the output of another DeNet simulator. Thus because it is more general and because it is more difficult to parse, we will use the model from Figure 3 in this example. NoDoSE, however, can work with either.

```

interface NodeParams {
attribute int node_number;
attribute string node_name;
attribute List<Struct OneParam {string name, string value}> parameters;
}
interface NodeResults {
attribute int node_number;
attribute string node_name;
attribute List<Struct OneResult {string name, real average, real std, int num}> results;
}
interface SimulationRun {
attribute String timestamp;
attribute List<NodeParams> node_params;
attribute List<NodeResults> node_results;
}

```

Figure 3: Example schema for the simulation output.

2.2 Decomposing the documents

The decomposition process begins by loading a single document into NoDoSE. The user then hierarchically decomposes the document using a GUI. Next additional documents of the same type are loaded in to the system and automatically parsed. Any errors are corrected by using the GUI and reparsing. The process is complete when all of the documents have been successfully parsed.

The first step in decomposing a document is indicating its top level structure, in this case a record of type `SimulationRun`. Next, we add each of its three fields (`timestamp`, `node_params`, and `node_results`) by selecting the relevant portion of the text in the document window and clicking on the add structure button in the tool bar (Figure 4). The type, type name, and label of each field can be entered using the controls on the bottom portion of the window. Since `node_params` and `node_results` fields are complex types (lists), the decomposition process must continue.

Suppose the user chooses to decompose the list of node results next. Double-clicking on that node in the tree view panel will display only the portion of the document mapped to the `node_results` list. The user then selects the text of the first element of the list (the first two lines) and adds this as a structure. Next, the second element of the list is added. Figure 4 shows a snapshot of the interface at this point. The user could continue to add every element in this manner but this would become tedious if there are many elements. Instead, the user can ask NoDoSE to try to infer the remaining elements by mining the text. If the tool mistakenly identifies elements the user can correct a few of the errors and ask that the text be reminded. In this way, the correct grammar for the component will eventually be learned. Once it is, NoDoSE is able to identify all of the other elements of the list correctly.

The decomposition process must continue since each element of the node results list is a record of type `NodeResults`. Any of the list elements can be selected and its fields added. Figure 5 shows the screen after the third element has been decomposed. As before, the user does not have to decompose every element by hand. Once a few elements (in this case, one) have been decomposed, the miner

can be again invoked to decompose every record of type `NodeResults`. Since the `parameters` field of `NodeResults` is itself a list, the process must continue (`node_params` must also be decomposed). The process continues until all of the leaves of the document tree are atomic types.

After the grammar for a particular file has been determined, NoDoSE is loaded with all of the other files of the same type. These are automatically parsed using the grammar inferred from the first file. It's possible, though, that parsing fails on one of the additional files for one of two reasons. First, the additional file may contain an error such as a mistyped field name or an OCR error. In such a case, the user can correct the error through the GUI but the grammar does not need to be updated. The second reason why parsing may fail is that the additional files contain something that was not present in the first file parsed. For example, suppose that the files come from two different versions of the simulator and that the newer version measures and outputs an additional value. If a file from the old version was used for the initial parsing process, NoDoSE will fail to recognize the new field. In this case, the user must correct the parsed tree for the new file, describing the new field using the GUI as before. The extractor will then update its grammar to account for the new field. After this, any of the files with the newly described field will be automatically reparsed. When all of the files of the same document type have been successfully parsed, the first step of the conversion process is complete.

2.3 Outputting the extracted data

The final step of the conversion process is to specify how to output the data that has been extracted from the parsed files. As shown in Figure 1, different options are supported. One option is to write the data into a text file. The format of the file and which data to be output is specified using a simple GUI-based report generator. The intent of this component is not to replace the querying and reporting functions of a DBMS but to provide a quick means of writing simple files, such as comma or tab delimited tabular data for input to spreadsheets and the like. For users who need to perform more complex operations on the data, NoDoSE can generate a schema file and a load file for use by a load utility provided by a third party DBMS. At the present, the generated schema file is ODL-like and the load file is in a generic format of our design. Additional formats can be added either by using the report generator or by coding an additional report component.

3 System Architecture

This section describes the internal architecture of NoDoSE in two parts: it first describes how the structure of documents is represented and then gives an overview of the components that comprise the system.

3.1 Document Model

Externally, documents are represented as flat files which serve as the input to NoDoSE. Internally, however, we need to be able to store information about the structure of the documents. Hence for every file that is loaded by the user, NoDoSE maintains a tree that maps the structural elements of the document to the text of the file (Figure 6(a)). Each node of the tree represents one of the

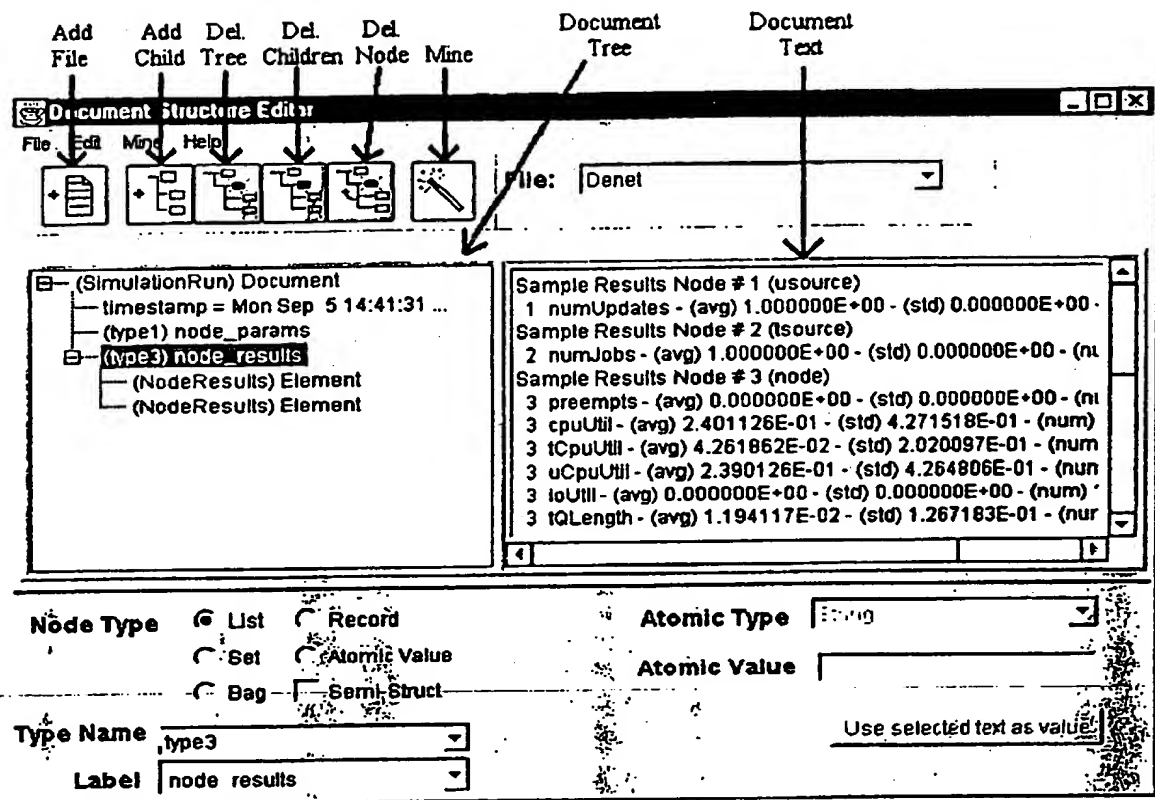


Figure 4: Screen shot of NoDoSE after a few steps.

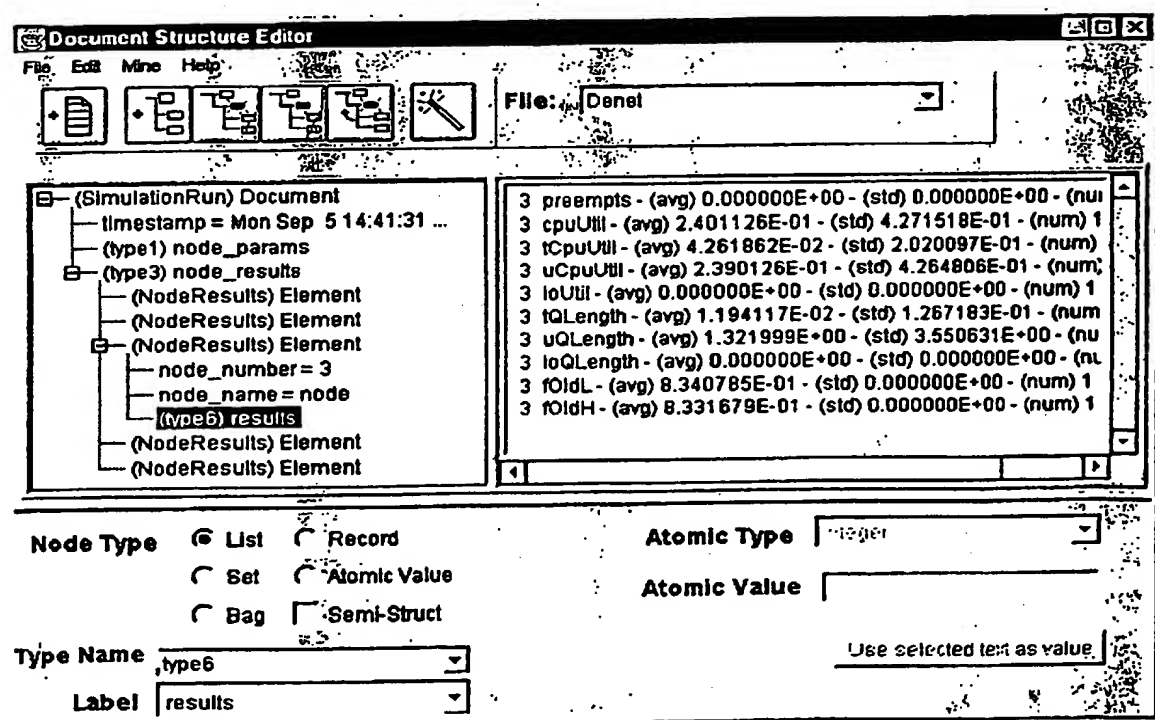


Figure 5: Screen shot of NoDoSE later in the process.

structural components of the document such as an element of a list or a field in a record. The following values are stored in each node:

typeName - Every node in the tree, and thus every component of a document, must be of either an atomic type or a named composite type. Details of the type system supported by NoDoSE are described below.

startOffset, endOffset - These two values indicate which portion of the file corresponds to the structural component. For non-root nodes, the offsets are relative to the start of the parent node's region.

label - The only required use of this field is in the children of record nodes to indicate which field the node represents. Labels can also be used to represent data in a schema-less model such as OEM [CGMH+97].

authorId - This identifies the creator of the node, the user or one of the mining components. Maintaining the originator of a node is useful when mining structure since user identified regions can usually be given greater credence than regions identified by the mining components.

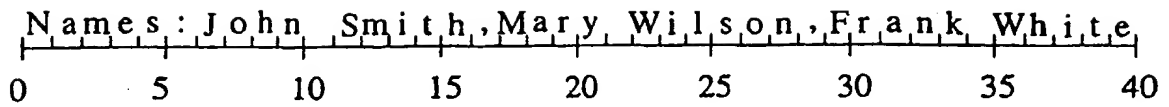
confidenceValue - This is a value between 0 and 1 indicating how confident the author is that this node is correct. It is typically set to 1, meaning complete confidence, for nodes added by the user and is set to a lower value for nodes inferred by one of the mining components. One practical use of the confidence value is to alert the user about nodes that may not have been parsed correctly (see Section 4.1.1).

To clarify the mapping process, consider the file shown in Figure 6(a) that is composed of just a single line. We can view this file as a list of names, each name being composed of a first and last name. This structure would be represented by the tree shown in Figure 6(b). The root of the tree represents the whole document and is mapped to the entire file. The root is of type *Doc* which is a list of objects of type *Name*. The root has three children, each corresponding to one of the names in the list, and in the same order as the corresponding names appear in the document. Each child is of type *Name*, which is a structure with fields for the first and last name. Of course each node has a different pair of offset values to indicate where in the text the element is.

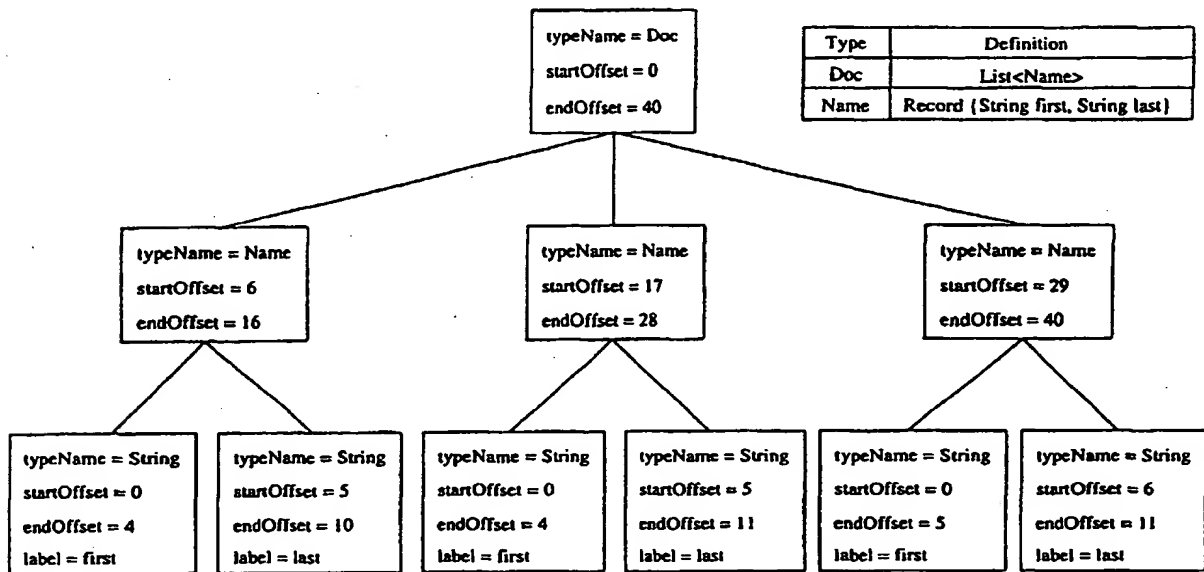
Finally, each node of type *Name* has two children corresponding to the two fields in the record. Each child is of type *String* which is atomic and hence they have no children themselves. Also, each child node has a label that identifies which field of the parent node it contains the value for. Note that all of the *firstName* nodes have a start offset of 0. This is because offset are relative to the parent node's text (and, in this case, the first name value begins every list element).

Every node in a document tree must be associated with a particular type. NoDoSE predefines six atomic types: *Integer*, *Float*, *String*, *Date*, *EmailAddress*, and *URL*. Additional atomic types can also be added — the user need only supply a name. Complex types can be defined as well using the following common type constructors:

1. *NewType* = *Set*<*OldType*>,
2. *NewType* = *Bag*<*OldType*>,
3. *NewType* = *List*<*OldType*>,



(a) Example file.



(b) Document tree for example file.

Figure 6: Representation of a document.

4. $NewType = Record\{OldType_1\ fieldName_1, OldType_2\ fieldName_2, \dots\}$.

Note that unlike some type systems, only singly nested types can be defined in a single step. Thus defining the new type $List<Record\{String\ first, String\ last\}>$ would require two new types, one for the record and one for the list.

In addition to the structured type constructors, NoDoSE provides an analogous set of type constructors for semistructured data: $SemiSet<>$, $SemiBag<>$, $SemiList<>$, and $SemiRecord\{fieldName_1, fieldName_2, \dots\}$. These constructors do not restrict the type of their components so, for example, all of the elements of the list do not have to be of the same type. Note that many of the previously proposed models for semistructured data do not require all four constructors. For instance, OEM[CGMH⁺97] objects can be represented using only atomic types and $SemiList$. We have added them, though, for cases where more semantic information is known.

Having covered the type system we can now define what constitutes a legal document tree. For a tree to be legal all of its nodes must be legal. A node n is legal if and only if all of the following conditions hold, the first four of which are related to type restrictions and the last of which is related to mapping:

1. if n is an instance of an atomic type it cannot have any children;

2. if n is an instance of a structured collection (List, Bag, or Set) all of the children of n must have the same type;
3. if n is an instance of a Record type defined as the set of fields $F = \{ \langle t_1, f_1 \rangle, \langle t_2, f_2 \rangle, \dots, \langle t_m, f_m \rangle \}$ and n has the children c_1, c_2, \dots, c_k :
 - (a) $(\forall i)[\langle c_i.typeName, c_i.label \rangle \in F]$,
 - (b) $(\forall i, j)[(1 \leq i \leq k) \wedge (1 \leq j \leq k) \wedge (c_i.fieldName = c_j.fieldName) \rightarrow (i = j)]$.
4. if n is an instance of a SemiRecord type defined as the set of fields $F = \{f_1, f_2, \dots, f_m\}$ and n has the children c_1, c_2, \dots, c_k :
 - (a) $(\forall i)[\langle c_i.label \rangle \in F]$,
 - (b) $(\forall i, j)[(1 \leq i \leq k) \wedge (1 \leq j \leq k) \wedge (c_i.label = c_j.label) \rightarrow (i = j)]$.
5. let p be the parent of n , l its left sibling, and r its right sibling. The following must hold:
 - (a) $0 \leq n.startOffset < n.endOffset \leq parentLength$ where $parentLength$ is taken to be $p.endOffset - p.startOffset$ if p exists and the length of the document otherwise;
 - (b) if l exists, $l.endOffset \leq n.startOffset$;
 - (c) if r exists, $n.endOffset \leq r.startOffset$.

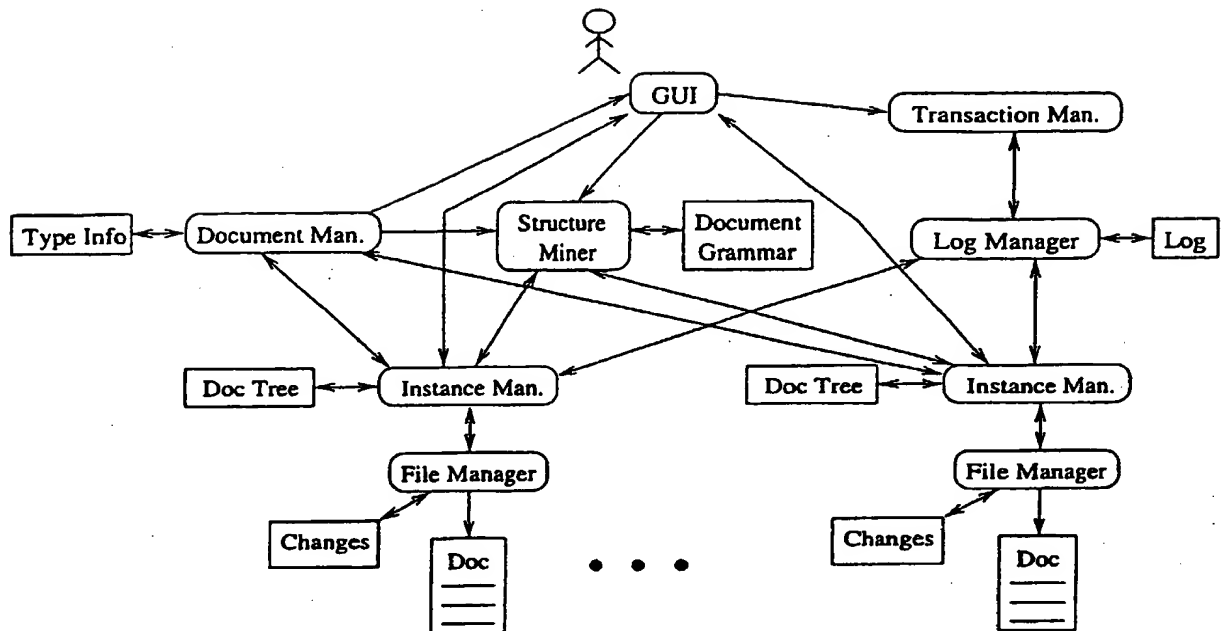
It is the responsibility of the *Instance Manager* and *Document Manager*, described below, to ensure that every tree instance is legal.

3.2 Components

NoDoSE is intended to be a test bed for studying the data extraction problem. Thus, rather than build a monolithic tool and force other researchers to wade through thousands of lines of source code, we've designed the system as a set of components that communicate through Java interfaces. Any of the components can be replaced independent of the others, and for certain types of components, more than one can be instantiated at any given time. At the present, changing components still involves changing a few lines of code in the top level NoDoSE class but with in the next version we plan to use the Java Reflection class to allow components to be changed or added dynamically without any code changes.

Figure 7 shows most of the components in NoDoSE and how they interact. The reporting component has not been shown in the interest of readability but will be discussed below. Most of the components provide the basic infrastructure for the system: reading files, maintaining document trees and type information, and supporting undo/redo. We expect that these components, which we collectively call the *support components*, will rarely be the subject of experimentation. The remaining three components are those most likely to be changed: the structure miners, the report generators, and the GUI.

Allowing third parties to build components that modify the data is dangerous since they may accidentally violate constraints. For example, a GUI may allow the user to add a child to an atomic type. To avoid these problems we have adopted the model-view-controller [Gol90,KGP88] paradigm. The model, which stores the data and enforces constraints, is maintained by the support



components. The other three components types, Reporters, Miners, and GUIs, all provide views of the data in the model. In addition, the miner and GUI both serve as controllers since they modify the data. This must be done through the model, however, which guarantees that no constraints can be violated. If a controller performs an action that would violate a constraint, an exception is raised by the model which can be trapped and handled by the controller.

Because there can be many views on the model, the core components also support the observer/observable paradigm [GEH⁺94]. For example, the type manager allows other components to register as observers of a particular type. Whenever a new instance is added to that type or an instance deleted, the observers are notified. The mining component described in Section 4 uses this notification to incrementally maintain its statistics about a given type.

Below we describe all of the components of the system. Details of the version 1.0 implementation can be found in Section 5.

File Manager - Enables sections of a file to be read or modified by the other components. Modifications are not performed directly on the original file; a separate file of changes is maintained. This ensures that NoDoSE cannot corrupt an input file and that it can work with read-only files or files residing on remote machines. There is a one to one mapping between file managers and files.

Instance Manager - Maintains the document tree for a file, providing all of the basic tree manipulation operations, such as node insertion and deletion. It also provides methods that map the tree to the file or vice versa. For example, when a user double clicks in the document text panel, the GUI can use the instance manager to find the tightest bounding node for the point in the file so that it can display its type information. A particular instance manager stores only a single tree so every file must have its own.

Document Manager - Maintains information about a class of document (i.e. the output of a particular simulator). In particular, it stores a list of all of the files of a particular document class, as well as information on all of the types used in the files. The document manager stores four pieces of information for each type in the system: its name, its definition, a list of all of the nodes of the type, and a list of *observers*, which are the components that want to be notified when any of the type information changes.

Log Manager/Transaction Manager - Supports user undo/redo.

Structure Miner - Attempts to automatically determine how to parse a given node type. This component is described in much more detail in Section 4.

GUI - Unlike in many systems, the GUI in NoDoSE is truly a replaceable component. In fact, we are currently working on an alternate user interface for structured documents through which the user first specifies the schema of the document in ODL and then maps the text of documents to the schema. In a sense, this is the opposite of the current approach.

Reporter - Outputs extracted information, usually as a report, a load file, or as information needed by a wrapper generator. Multiple Reporter components can be active within one system to give the user different output options.

4 Mining for Structure

This section describes the two mining/parsing components that have been implemented so far: one that mines text files and one that parses HTML code. Both components are limited in scope; The text miner only handles structured types and the HTML parser does not handle frames or other advanced features. Despite their limitations, however, we have been able to extract data from an interesting set of documents. Further, building two different mining components has forced us to ensure that the interfaces exposed to the mining components are powerful enough and clean enough to support different algorithms. Details of both mining components appear below.

4.1 Plain text miner

The component described in this section attempts to determine the parsing rule for instances of a type. The particular type being mined in any invocation is called the *target type*. For this first version of NoDoSE, we chose to concentrate on mining structured types (Set, Bag, List and Record) since the type of the children nodes are known by definition. After we develop robust algorithms for this mining problem we plan to study the mining of semistructured types.

Another simplification in the current version is that we use the same algorithm for all of the collection types (Set, Bag, and List) since the semantic differences between them are rarely noticeable at the level of the format of the text file. Thus, for the remainder of the section, when we discuss mining Lists our comments will be equally valid for mining any collection type.

The algorithms for mining lists and records are both based on the same overall three step strategy:

<i>Meaning</i>	<i>Notation</i>
Begin with marker	[With Marker "marker"]
Begin after marker	[After Marker "marker"]
Begin at fixed offset	[Offset offset]
End with marker	[With Marker "marker"]
End before marker	[Before Marker "marker"]
End after a fixed # of lines	[After Lines num_lines]
End at fixed offset	[Offset offset]

Table 1: Parse rule components for the plain text miner.

1. **Theory generation** - Create a set of theories for how to parse the instances of the target type. For a list type, a simple theory is "each element will be separated by a comma".
2. **Theory evaluation** - For each theory under consideration,
 - (a) Parse every node that is an instance of the target type (a list of which can be retrieved from the Document Manager) to generate a list of predicted children nodes. Note that if multiple documents are loaded, not all of the nodes will necessarily be from the same document.
 - (b) Compare the predicted nodes to the nodes that are actually present in the document tree. Count the number of nodes in the document trees that were not predicted, which we call *false negatives*, and the number of predicted nodes that cannot possibly be correct, which we call *false positives*.
3. **Theory application** - If one or more of the theories has no false positives or negatives, pick one of them and add its predicted nodes to the document tree (or trees, if more than one document is loaded).

Although we will need different types of theories for parsing lists and records, the two share common elements: In each case, we are trying to subdivide the portion of the file corresponding to a given node, which we call the *node text*, into smaller units, each either a list element or a record field. To find the boundaries of the units we will need two theories: a theory about how the beginning of a unit is determined, called a *start theory*, and a theory about how the end of a unit is determined, called an *end theory*. We call the combination of a start theory and an end theory a *unit theory*.

Table 1 lists all of the start and end theories used in the current implementation of NoDoSE's text mining component. Most include a variable that must be instantiated, often a marker which is a string that separates units. For example, in the file from Figure 6, the best start theory would be [After Marker ","]. We could use something similar for the best end theory: [Before Marker ","]. To represent the resultant unit theory, which is the combination of the start and end theory, we will write <[Before Marker ","],[Before Marker ","]>. Rather than explain the meaning of the rest of the theories here, we introduce the concrete problem of parsing lists to give the discussion more context.

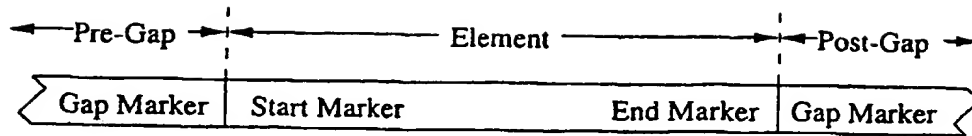


Figure 8: Presumed format of a list element.

4.1.1 Mining lists

Mining a list type entails two tasks: find a parsing rule that identifies every element of a list of the target type and then parsing every instance of the target type using the rule. Figure 9 shows the three instances of a target type, $\text{Roster} = \text{List}\langle\text{String}\rangle$, that will serve as a running example for this section. The lists is meant to represent the rosters of basketball teams². The boxes around parts of the lists indicate the elements that the user has already identified.

The mining algorithm for list types depends on three assumptions:

1. Every element of the list will have the same type. Since this algorithm will only run on structured collection types, this assumption must hold.
2. Every element of the list will have the same format. This assumption does not necessarily hold but seems reasonable given assumption 1 and simplifies the grammar induction.
3. If k elements of a list have been identified by the user, they will be the first k elements in the list. This is a very powerful assumption because it gives the miner a way to identify theories that generate false positives — no predicted unit can appear before any preexisting unit in a list. It does, however, impose restrictions on how structure information must be input by the user.

Of course all three assumptions hold for our example. The first holds by definition: all of the elements are of type `String`. The second holds as well since each element has the basic format "Player Name: *name*". The third also holds since list 1 has all of its elements specified, list 2 has none of its elements specified, and list 3 has only one element specified but its the first element. An example of a violation of assumption 3 would be if list 2 had the player named Hill specified without having Dumars specified as well.

Lists can be viewed in general terms as a header followed by the elements of the list separated by gaps. Of course, specific list types may not have headers or gaps at all. Figure 10 shows how list 1 of our example fits this pattern. For any list with at least one element defined, we know the boundaries of its header — everything to the left of the first element by assumption 3 above. Also, everything between two defined elements is necessarily a gap since no other element could exist between the two according to assumption 3. Thus even if some of the list instances have no elements defined and others have only some of their elements defined, the miner will usually still be able to identify a few headers and gaps (if the list text has them). In our example, the headers of lists 1 and 3 are known as is the gap between elements 1 and 2 in list 1.

²If the user were interested in capturing the name of the team as well he would probably not define the example strings to be Lists at all. Instead, they could be defined (in two steps) as `Record{String teamName, List<String> playerNames}`.

List 1 → Team Name: Hawks; Name: Anderson Name: Blaylock
 List 2 → Team Name: Pistons; Name: Dumars Name: Hill Name: Hunter
 List 3 → Team Name: Bucks; Name: Allen Name: Brandon

Figure 9: Example lists.

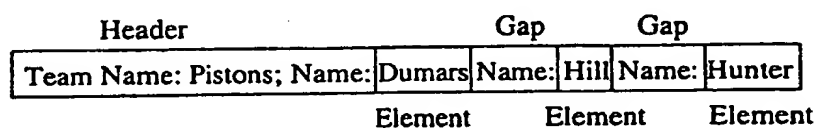


Figure 10: Presumed format of a list.

The task of the miner is to generalize from these examples to discover how to identify the headers and gaps in all of the lists. This will require two types of theory: an end theory for finding the end of the header (if present), which we will call the *header theory*, and a unit theory for finding the beginning and end of the elements of the list, which we will call the *element theory*.

Simplified psuedo-code for mining algorithm used in NoDoSE is shown in Figure 11. The code employs the three steps described in the preceding section: theory generation (lines 1-2), theory evaluation (lines 3-28), and theory application (lines 29-32). Each of the steps is discussed in detail below.

Theory generation For lists, we have two types of theories to generate, header and element theories. Let us first consider how to generate the set of header theories, T_H . We begin with the end theories from Table 1. Next, we instantiate the placeholders in the theories which means choosing markers, offsets, and numbers of lines. For example, consider instantiating the marker in the theory [With Marker "marker"]. To do so, we need to examine all of the known headers and find their longest common suffix. In the running example, the longest common suffix is the string "; Name: " and so the instantiated theory, [With Marker "; Name: "] is added to T_H . (In fact, the prototype actually adds two almost identical version of this theory, one in which the marker is case sensitive and one in which it is not, but this does not affect the algorithms in any fundamental way and hence will not be further mentioned.)

Often there will not be any consistent value with which to instantiate a theory. For instance, there may be no common suffix at all or the headers may not all have the same number of lines in them. In this case, the theory is not added to H_T .

Generating element theories is similar to generating header theories except that element theories are composed of both a start theory and an end theory — header theories do not require start theories since they always begin at offset 0. With the possible exception of the first and last elements in a list, an element can be viewed as shown in Figure 10. A pre-gap is the gap between the element and the preceding element and a post-gap the gap between the element and the following element. The theories for parsing list elements are based on trying to find a common suffix in the pre-gaps, a common prefix in the elements, a common suffix in the elements, or a common prefix in the post-gaps. In the example lists, the common prefix and suffix of the gaps are both "Name: "

and the elements do not have a common start or end marker. Thus the set of valid start theories, T_{start} , is {[After Marker " Name: "]} and the set of valid end theories, T_{end} is {[Before Marker " Name: "]}]. The set of candidate element theories, T_E , is computed as $T_{start} \times T_{end}$, which in this case is just {[After Marker " Name: "],[Before Marker " Name: "]}].

Theory evaluation This step represents the majority of the code in Figure 11. Conceptually, the code chooses the best header theory and then uses that theory in trying to find the best unit theory. This is a bit of a short cut since the algorithm should really consider all possible pairs of header and element theories. Unfortunately, this is very expensive with even a moderate number of different theories. Hence we choose to separate the two tasks, realizing that in some cases we may miss the best pairing.

The best header theory is determined in Lines 3 through 10. The algorithm tries each theory, using it to predict the headers of all of the list instances. For those lists that have at least one element defined, the header is known and can thus be checked against the predicted header. If one of the theories correctly identifies all of headers it is assumed to be correct and saved for use in element parsing. If no theory is correct, we assume that headers do not have to be specially handled for the target list type.

In the example from Figure 9, it is critical that the headers are handled. By skipping past the header, the element theory <[After Marker " Name: "],[Before Marker " Name: "]> can be used to correctly identify every element. If the header is not skipped, the same element theory will predict that the first element of the list starts with the " Name: " marker that is part of "Team Name: " and thus mining will fail. If the headers were "Team: " instead of "Team Name: ", however, the element theory would work even if headers were not skipped. Thus many lists do not require a header to be found at all and therefore the mining algorithm continues even if no consistent header theory can be found.

The code from Lines 11 to 28 uses each element theory to predict the elements in all of the lists. For each list, the search for elements starts at the first character in the unit text unless a header is present in which case the search begins immediately after it (line 17). The start theory is used to find the predicted beginning of the next element (line 19), and if its successful, the end theory is used to find the predicted ending of the element (line 21). If a new element is found it is added to the predicted set. The search continues in the same unit text until no more elements are found. At this point the predicted elements are compared again the elements defined by the user. The function *findFalseNegatives(predictedSet, actualSet)* counts the number of user defined elements that were not predicted by the theory, $|actualSet - predictedSet|$. The function *findFalsePositives(predictedSet, actualSet)* counts the number of predicted elements that must be incorrect, which are the new predicted elements that start before the last user defined element ends (by assumption 3). For example, in Figure 9 if a theory predicted that an element other than "Allen" started prior to the space after "Allen" in list 3, it would have to be incorrect. For list 2 which has no user defined elements, however, no predicted element can be eliminated.

Theory application As long as at least one consistent element theory has been found, the elements that it predicted and that were not in the original document trees are added using the Instance Manager (lines 29-32). The new elements are of the same type as all of the other children

```

(1) Create and instantiate the set of header theories,  $T_H$ .
(2) Create and instantiate the set of element theories,  $T_E$ .
(3)  $t_{h_{best}} = \text{null}$ 
(4) for each header theory,  $t_h$ , in  $T_H$ 
(5)   for each list  $l$  in  $L$ 
(6)      $offset = \text{findHeaderEnd}(t_h, l)$ 
(7)     if ( $l$  has a header and  $offset \neq l.\text{headerEnd}$ ) then
(8)        $t_h.\text{errors}++$ 
(9)     if ( $t_h.\text{errors} == 0$ ) then
(10)       $t_{h_{best}} = t_h$ 
(11)  $t_{e_{best}} = \text{null}$ 
(12) for each element theory  $t_e$  in  $T_E$ 
(13)   for each list  $l$  in  $L$ 
(14)      $offset = 0$ 
(15)      $predicted = \{\}$ 
(16)     if ( $t_{h_{best}} \neq \text{null}$ ) then
(17)        $offset = \text{findHeaderEnd}(t_{h_{best}}, l)$ 
(18)       while ( $offset < l.\text{length}$ )
(19)          $start = offset = \text{findElementStart}(t_e, l, offset)$ 
(20)         if ( $start \neq -1$ )
(21)            $end = offset = \text{findElementEnd}(t_e, l, offset)$ 
(22)           if ( $end \neq -1$ )
(23)              $predicted = predicted \cup \{< l, start, end >\}$ 
(24)            $t_e.\text{predicted} = t_e.\text{predicted} \cup predicted$ 
(25)            $t_e.\text{falsePos} += \text{findFalsePositives}(predicted, l.\text{elements})$ 
(26)            $t_e.\text{falseNeg} += \text{findFalseNegatives}(predicted, l.\text{elements})$ 
(27)         if ( $t_e.\text{falsePos} == 0 \wedge t_e.\text{falseNeg} == 0$ ) then
(28)            $t_{e_{best}} = t_e$ 
(29)       if ( $t_{e_{best}} \neq \text{null}$ ) then
(30)         For each element  $< l, start, end >$  in  $t_{e_{best}}.\text{predicted}$ 
(31)           if  $< start, end > \notin l.\text{elements}$ 
(32)              $l.\text{elements} = l.\text{elements} \cup \{< start, end >\}$ 

```

Figure 11: The (simplified) algorithm for mining and parsing list types.

of the target type, the author ids identify the text miner as the author, and a confidence factor can be assigned. In this version the confidence factor of the new elements is set to 0.5, but in the future we plan to compare each element against statistics gathered on all of the elements to try to identify questionable predictions. For instance, if all of the elements are 40 characters long except for one which is 80, it is likely that due to an error in the parsing rule or a typo in the document, two consecutive elements have mistakenly been parsed as one.

4.1.2 Mining records

The mining algorithm for record fields depends on four assumptions:

1. Every field in a record has a unique name. This assumption is enforced by the Instance Manager.
2. If the fields of two different records of the same type have the same name, the two fields themselves will have the same type. Such fields are called *corresponding* fields. For example, if two record of the same type both have fields named *phoneNumber*, the two *phoneNumber* fields should have the same type. This assumption is also enforced by the Instance Manager.
3. All corresponding fields will have the same format. This assumption is not forced on the mining component but it seems reasonable given assumption 2 and simplifies the grammar induction.
4. The fields in a record instance are either completely identified by the user or not identified at all. Thus if k fields of a record instance have been identified by the user, they will be the only k fields in that instance. This is a very powerful assumption because it gives the miner a way to detect a parsing theory that generates false positives — a predicted field must appear in a record if the user has identified any fields at all in that record.

Note the assumptions that this component does *not* make: every field is present in every record instance, and the order of fields within a record is fixed. Thus we are able to parse a limited but useful class of semistructured documents.

Mining lists and records are similar except for one important difference. For lists, we assume that the format of every element is the same. This assumption allows the list mining algorithm to consider only two sets of theories, one to skip past the header and one to identify elements. In contrast, every field in a record type may have a different format and thus every field requires its own set of theories. Further, the order in which the algorithm tries to parse the fields is important.

For example, consider the text of a record that contains, among other things, the string "Name: Smith, John.". Suppose the user chooses to model a name as two fields, *LastName* and *FirstName*. The best theory for identifying a *LastName* field might be $\langle [\text{After Marker "Name:"}], [\text{Before Marker ","}] \rangle$. If we know that a *FirstName* field always follows *LastName*, its rule would be $\langle [\text{After Marker ","}], [\text{Before Marker "."}] \rangle$. We must be careful, though, to only try to apply the unit theory for a first name immediately after parsing a last name. Otherwise, an unrelated comma anywhere in the text of the record would lead to the first name being falsely parsed.

To avoid problems of this sort, the mining algorithm tries to find an order for the fields in a record type that is consistent across all of its instances. This is difficult for two reasons:

```

(1) Pick a consistent order for the fields of the target type.
(2) numFieldsMined = 0
(3) for i = 1 TO Nf
(4)   Create and instantiate the set of field theories, TFi.
(5)   tbesti = null
(6)   for j = 1 TO Nr
(7)     lastOffsetj = 0
(8)   for i = 1 TO Nf
      — Try to find field i in every record —
(9)     for j = 1 TO Nr
(10)       for each unit theory t in TFi
(11)         start = findFieldStart(t, rj, lastOffsetj)
(12)         if (start ≠ -1) then
(13)           end = findFieldEnd(t, rj, start)
(14)           if (end ≠ -1)
(15)             t.predicted = t.predicted ∪ {< rj, start, end >}
(16)             t.falsePos += findFalsePositives({< rj, start, end >}, rj.fields)
(17)             if (start = -1 ∨ end = -1) then
(18)               t.falseNeg += findFalseNegatives(i, rj.fields)
      — Find the first theory that perfectly predicted field i using the theories —
(19)     for each theory t in TFi
(20)       if (t.falsePos == 0 ∧ t.falseNeg == 0) then
(21)         tbesti = t
(22)         numFieldsMined ++
(23)         for j = 1 TO Nr
(24)           newOffsetj = findFieldEnd(t, rj, findFieldStart(t, rj, lastOffsetj))
(25)           if (newOffsetj ≠ -1) then
(26)             lastOffsetj = newOffsetj
(27)         break
(28)   if (numFieldsMined = Nf) then
(29)     for i = 1 TO Nf
(30)       For each element < r, start, end > in tbesti.predicted
(31)         if (< start, end > ∉ r.elements)
(32)           r.elements = r.elements ∪ {< start, end >}

```

Figure 12: The simplified algorithm for mining and parsing record types.

1. Not all fields are present in each record instance and no single instance is guaranteed to have every field in it. Thus we may have to look at more than one record instance to determine the field order and may not be able to determine a unique ordering even if we look at every record instance.
2. An important ordering may only exist between subsets of the fields in the record and the other fields may exhibit an inconsistent ordering.

The miner uses a simple algorithm that computes for each field, the set of all fields that have preceded it in at least one record and the set of all fields that have followed it in at least one record. The two sets can be used to find a totally consistent ordering if one exists although it is not guaranteed to handle the second complication from above. So far, this has not been a problem in the documents we have mined.

The psuedo-code for the record mining algorithm is shown in Figure 12. Lines 1 through 7 do theory creation and general initialization. Theory variables are instantiated by comparing the corresponding fields in all of the records, looking for a common pre-gap marker, start marker, end marker, or post-gap marker as was done with list elements.

The for loop starting at line 8 performs the meat of the algorithm: All of the record instances of the target type are parsed in parallel, one field at a time. Each theory for the current field number is tried to see if it can identify the field in each of the record instances (lines 11-15). If so, the algorithm calls *findFalsePositive* to see if the field should really have been found. Using assumption 4, a predicted field is a false positive if it has not already been defined and the user has defined at least one field in the record instance. If no field was found for a record instance, the algorithm checks that none was defined by the user (lines 17-18).

After all of the theories have been tried on all of the record instances, the first consistent theory is chosen (lines 19-26). In addition, the current offsets into all of the records are updated to account for the newly parsed fields. The main loop then repeats, starting from the new offsets and looking for the next field.

After all of the fields have been parsed, the algorithm checks that it has found a consistent theory for every field. If it has, all of the fields predicted by the consistent theories are added to their records.

4.2 HTML Parser

The HTML parser available as part of NoDoSE parses documents or subdocuments based completely on structural information using recursive descent. Unlike the plain text miner, the HTML parser does not store any internal information about a type since it parses based on the static grammar rules of HTML. The tags that are understood by the parser are listed in Table 2 with a brief description (due to space constraints) of how they are represented in the document tree. Other tags are just considered part of the text of the document. The following discussion is necessarily brief due to space constraints.

In practice, the HTML parser generates more structure than the user is interested in (i.e., information from meta tags). After parsing, he should delete the nodes that are not of interest and rename types and labels to be semantically meaningful. The changes are not recorded by the HTML parser, though, so the next instance of the same page type will require all the changes to

<i>Tags</i>	<i>Representation</i>
head,title,meta	Head is represented as a Record with two fields: title, which is a String extracted from the title tag, and meta, which is a List of strings extracted from the meta tags.
body	Body is represented as a SemiList. Typically its children will be first level headings.
h1,h2,h3,h4,h5,h6	Each heading is represented as a SemiList. Typically its children are paragraphs, represented as Strings, and sub-headings.
ul,ol,dir menu,li	All of the list formats are represented by a SemiList of list items. Lists can be arbitrarily nested.
table,tr,td	A table is represented as a SemiList of rows. Each row is a SemiList of the data in the cells.
anything else	Represented in the tree as a string.

Table 2: Translation of HTML tags into document structure.

be made again. To avoid this problem, the plain text miner can be run on the parsed document produced by the HTML miner. It will try to infer the format of the file just as if the structure had been entered using the GUI. If it is successful, future HTML files of the same type can be parsed automatically.

The current version of the plain text parser only works with structured types, however, so additional modification of the document tree produced by the HTML parser is necessary. Thus the typical use of the HTML parser follows these steps:

1. Use the GUI and plain text miner to decompose the document until the HTML portion is reached. If the entire document is HTML, this step can be skipped.
2. Run the HTML parser on the HTML portion. This creates a sub-tree below the current node that represents the HTML portion of the document parsed.
3. Use the editing capabilities of the GUI to convert the sub-tree to a form suitable for mining. For example, the user must delete the nodes that are not of interest and rename types and labels to be semantically meaningful. Also, the nodes created by the HTML parser will all be semistructured (except for the atomic types). The types of these nodes must be changed to a structured type before the miner described in the previous section can work on them.
4. If there are other documents of the same type, load them and use the plain text miner to mine them.
5. Perform any needed edit steps and repeat step 4 until the plain text miner get the correct results for all of the files.

The approach of running the miner on the tree produced by the parser is an attractive one since it lets the text miner benefit from the knowledge of the document syntax without requiring that the parser and miner communicate directly. The same approach can be used on other files types with known syntax such as latex files or mail files. For example, consider a mail file consisting of responses to a survey. Each message will have all of the normal header information plus a highly formatted message body with the survey answers. By parsing the file using the mail syntax and

then mining the message bodies to structure the survey responses, a user can quickly export the survey data along with header data (send, time it was sent) to a DBMS.

5 Implementation

A prototype that implements a subset of the components described in Section 3 has been implemented in approximately 5000 lines of Java code. Two of the components, the Transaction Manager and the Log Manager, have not been implemented at all although the interfaces have been designed; thus NoDoSE version 1.0 does not support undo. Also, the current version only contains one Reporting component that writes the extracted data in a generic format similar to OEM. It also outputs an ODL schema for the data if it is not semistructured.

We have run NoDoSE on many different files, including simulator output, mail files, c source code, OCR'd documents, and many web pages. The results are difficult to quantify, although overall we have been pleased at the wide range of documents NoDoSE can extract data from. Part of this success, however, is the result of learning how to work around the quirks of the system. For example, the miner is currently very sensitive to where the user chooses the boundaries between list elements and record fields (i.e. whether to include the final carriage return in the selected text).

Also, the dependence of many of the theories on constant string markers causes problems. For instance, the records of type OneParam from the simulation output example (Figure 2) look like "5 misDL - (avg) 9.295315E-01 - (std) 2.559869E-01 - (num) 2455" where the first value, 5 in this case, is the simulator node number that the measured variable is in. This value is redundant and is thus not part of the record type we defined for OneParam. The consistent pre-gap marker for the variable name (misDL in this case) is two spaces since OneParam records from other nodes start with their own node number, i.e., " 4 ". Unfortunately, identifying the beginning of the variable name by two spaces will fail since the node number, which precedes the variable name in the text, is also preceded by two spaces, and will thus be falsely identified as the variable name. This problem can be avoided by including the node number in the record even though it is redundant. Doing so "eats" the node number so that two spaces serves as an adequate pre-gap marker for the variable name field. To obviate the need for such workarounds in the future we're more developing more flexible markers based on regular expressions.

The performance of the system is fine for small files — the mining wait is never more than a second or two. We have not been able to deal with large files, however, due to the state of Java. One problem is with the TextArea component which is used to display portions of the file on the right side of the program window. The component in the toolkit uses the windows peer which does not support files over 32kb and we have not been able to find a pure Java component with adequate performance for files over 100kb. Luckily, many interesting document types, especially web pages, are well within this limit so this restriction has not significantly hampered our research, although it has made measuring scalability impossible. Given the commercial push for Java, it's reasonable to believe that such problems will be corrected in the near future.

6 Related Work

NoDoSE makes two major contributions to the data extraction problem: its open architecture for structural mining and the plain text mining component that has been implemented in version 1.0 of the system. To our knowledge, the former has not been proposed by any other researchers and hence we do not discuss it further in this section. Instead, we concentrate on the approaches others have taken to the latter problem: mining structure and extracting data from documents.

The three efforts that are most closely related to our own are [AK97a,AK97b], [HGMC⁺97], and [KWD97]. The system built by Ashish and Knoblock [AK97a] is closest to NoDoSE in its approach: to infer the structure of a document by combining automatic analysis with user input. Their system is designed for web pages only: it uses font size information, HTML tags, and indentation to guess a page's structure. A user can then correct the guesses by instructing the system to ignore certain keywords and by identifying new keywords that the system missed. The advantage of this system is that certain types of pages can be parsed with very little user input since the system leverages its knowledge about HTML syntax and about how characteristics like font size are used to indicate nesting. The major disadvantages of the system is that because it depends on HTML tags, it is not useful for any other type of document. Also, it deals with only single instances of documents so it is unclear that it can be used in cases where no single instance of a document type has all of the features of the type.

Kushmerick, Weld, and Doorenbos describe a system [KWD97] that automatically extracts data from web pages although it will also work with plain text files. The extracted data must be representable as a set of tuples; no deep structure can be inferred. The advantage of the system is that no user interaction is required — the system infers the grammar of a document through a machine learning algorithm applied to many instances of the document type. The algorithm must be provided with domain knowledge, however, in the form of oracles that can identify interesting types of fields within a document. Further, if the algorithm fails, there is no information the user can provide to help it. The authors report a success rate of 48% on Internet information resources which is impressive for a fully automatic algorithm but not adequate for most applications. Still, their work contains interesting ideas for automatic parsing and their notion of corroborating recognizers is similar to way we evaluate theories against user input.

The final system we discuss, that of Hammer et. al. [HGMC⁺97], is unlike the others in that it is fully manual — the user must code a wrapper for their document type using a toolkit. The toolkit provides many constructs, especially for HTML processing, that make it easier than writing a parser directly in Lex and Yacc [Joh75]. It also provides the most control over the output format of the extracted data as well as the best support for semi-structured data. The obvious disadvantage is that the user must be able to analyze their documents and then code their wrapper which limits the usefulness of this approach as a rapid data integration tool. We are considering it, however, as one of the output formats of NoDoSE to give users more control over the output format of their data.

For comparison, the salient features of the three related projects described above, as well of those of NoDoSE, are summarized in Table 3. In comparison to the other systems, NoDoSE has two primary advantages:

1. It is the only system that can infer the structure of text files and has support for HTML

<i>System</i>	<i>Grammar Generation</i>	<i>Nested structure</i>	<i>Semi-structured data</i>	<i>Text documents</i>	<i>Support for HTML</i>	<i>Open Architecture</i>
Ashish & Knoblock	Semi-automatic	Yes	Somewhat	No	Yes	No
Hammer et. al.	Manual	Yes	Yes	Yes	Yes	N/A
Kushmerick & Weld & Doorenbos	Automatic	No	No	Yes	No	No
NoDoSE	Semi-automatic	Yes	Somewhat	Yes	Yes	Yes

Table 3: Comparison of different data extraction tools.

documents.

2. It is the only system that can serve as a test bed for structure extraction experiments since the mining components are well separated from the rest of the system.

We note that except for the system of Hammer et. al. (which does not have a mining component), there is no reason that the other mining algorithms could not be integrated as mining components in NoDoSE. This would yield improved handling of HTML documents or with portions of documents that contain HTML while retaining the plain text capabilities.

Finally, we mention two additional studies that are related to this work. First, in [DEW97] the authors built a system, ShopBot, for the automatic extraction of product and pricing information from on-line shopping web sites. The system performs reasonably well since it is able to leverage its domain knowledge about shopping but is not applicable to other domains and it was not considered in the above comparison. Second, in [AJ97] the authors describe a fully manual GUI-based tool for converting structured files from one format to another. Unfortunately, not enough information is provided to compare it to the studies described above.

7 Conclusions

Given the amount of interesting data that is in HTML pages or text files rather than in database systems, users have a strong need for a tool to extract data from such sources. This paper described a tool, NoDoSE, designed explicitly for these needs. NoDoSE serves two purposes. First, it provides a general architecture for the exploration of the data extraction problem, allowing other researchers to plug in their own mining algorithms, user interfaces, or report generators, without having to build the entire framework themselves. Second, it contains a component that is capable of inferring the structure of a useful class of text files, allowing data to be quickly extracted without coding.

The results of using NoDoSE on the type of documents we were originally targeting, simulation output and web pages, are promising. We have also noticed that many documents with complex parsing rules, such as files of c code, that should be beyond NoDoSE's reach are not due to stylistic conventions like indentation and standard comment blocks. On the other hand, we have occasionally been surprised to find very simple and very regular looking documents that NoDoSE cannot handle.

Usually, the failure is due to the dependence of many of the parsing theories on constant markers that delimit list elements or record fields. Thus we are currently developing an alternate approach based on regular expressions. We are also finishing the documentation of the component interfaces and will release NoDoSE over the web soon thereafter.

References

- [Abi97] S. Abiteboul. Querying semi-structured data. In *Proceedings of ICDT* (invited talk), 1997.
- [AJ97] A. Aiken and M. Jacoby. Data format conversion using pointer tree automata. Technical report, University of California, Berkeley, 1997.
- [AK97a] N. Ashish and C.A. Knoblock. Semi-automatic wrapper generation for internet information sources. In *Proceedings of cooperative information systems*, 1997.
- [AK97b] N. Ashish and C.A. Knoblock. Wrapper generation for semi-structured internet sources. In *Workshop on management of semistructured data*, 1997.
- [CGMH+97] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: integration of heterogeneous information sources. In *Proceedings of the meeting of the processing society of japan*, 1997.
- [DEW97] R. Doorenbos, O. Etzioni, and D.S. Weld. A scalable comparison-shopping agent for the world-wide web. In *Proceedings of the first international conference on autonomous agents*, 1997.
- [GEH+94] Gamma, Erich, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of object-oriented software architecture*. Addison-Wesley, 1994.
- [Go190] A. Goldberg. Information models, views, and controllers. *Dr. Dobb's Journal*, July 1990.
- [HGMC+97] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting semistructured information from the web. In *Workshop on management of semistructured data*, 1997.
- [Joh75] S.C. Johnson. Yacc — yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.
- [KGP88] Krasner, Glenn, and S. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-oriented programming*, August/September 1988.
- [KWD97] N. Kushmerick, D.S. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *Proceedings of IJCAI*, 1997.
- [Liv90] M. Livny. DeNet user's guide. Technical report, University of Wisconsin-Madison, 1990.

Semi-automatic Wrapper Generation for Internet Information Sources *

Naveen Ashish and Craig A. Knoblock
Information Sciences Institute and
Department of Computer Science
University of Southern California
4676 Admiralty Way, Marina del Rey, CA 90292
{ashish,knoblock}@isi.edu
<http://www.isi.edu/sims/{naveen,knoblock}>

Abstract

To simplify the task of obtaining information from the vast number of information sources that are available on the World Wide Web (WWW), we are building information mediators for extracting and integrating data from multiple Web sources. In a mediator based approach, wrappers are built around individual information sources to translate between the mediator query language and the individual sources. We present an approach for semi-automatically generating wrappers for structured internet sources. The key idea is to exploit formatting information in Web pages to hypothesize the underlying structure of a page. From this structure the system generates a wrapper that facilitates querying of a source and possibly integrating it with other sources. We demonstrate the ease with which we are able to build wrappers for a number of Web sources using our implemented wrapper generation toolkit.

1. Introduction

We are building information agents or mediators to gather and integrate information from multiple World Wide Web sources. The mediator [3, 18] approach has been used to integrate information from distributed heteroge-

neous database systems, where a mediator insulates the user from problems caused by different locations, query languages, and protocols of the different sources. We are extending the mediator approach to integrate information from multiple Web sources. Our approach is to take several related Web sources in a particular *domain* of interest (e.g., finance, government, or real-estate) and provide integrated access to multiple Web sources through a mediator.

For example, we can use a mediator to provide integrated access to multiple Web sources that provide information on countries in the world. An excellent Web source is the *CIA World Fact Book*,¹ which provides information on the geography, economy, government, etc., of every country. Other interesting sources include the Yahoo listing of countries by region from where we can obtain information such as what countries are in Europe, the Pacific Rim, etc. Another interesting source is the on-line listing of country corruption rankings. A user could query a mediator that provides access to the above sources to answer queries such as "Find the *Economic Overview*, *Telephone System* and *Corruption Rankings* of all countries in the Pacific Rim." The mediator would determine what sources can be used to answer the query, retrieve information from these sources, and present the integrated result to the user. There are several other research projects that are working on integrating Web-based sources. These projects include InfoSleuth [4], the OBSERVER project [15], the Information Manifold [11], and the Internet Softbot [7].

An essential component in a mediator architecture is a *wrapper* around each individual data source (see Figure 1), which accepts queries from the mediator, translates the query into the appropriate query for the individual source, performs any additional processing if necessary, and returns

* This work is supported in part by the University of Southern California Integrated Media Systems Center (IMSC) - a National Science Foundation Engineering Research Center, by the Rome Laboratory of the Air Force Systems Command and the Defense Advanced Research Projects Agency (DARPA) under contract number F30602-94-C-0210, by the National Science Foundation under grant number IRI-9313993, and by the DARPA Fort Huachuca Contract DABT63-96-C-0066. The views and conclusions contained in this paper are the authors' and should not be interpreted as representing the official opinion or policy of DARPA, RL, NSF or any person or agency connected with them.

¹ <http://www.odci.gov/cia/publications/nsolo/wfb-all.htm>

the results to the mediator. To make the integration of Web sources using the mediator approach feasible, wrappers are needed for all of the Web sources to be accessed. Wrappers for Web sources would accept a query from the mediator, fetch the relevant pages from that source, extract the requested information from the retrieved pages and return the results to the mediator. Essentially the wrappers make the Web sources look like databases that can be queried through the mediator's query language. The basic techniques applied in database integration using mediators can then be applied to Web sources integration. It is however impractical to construct wrappers for Web sources by hand for a number of reasons:

- The number of information sources of interest is very large, even within a particular domain.
- Newer sources of interest are added quite frequently on the Web.
- The format of existing sources often changes.

We report on the development of an implemented wrapper generation toolkit that provides a semi-automatic, interactive wrapper generation facility for Web sources. It should be noted that building wrappers is just one of the challenges in building the kinds of information mediators for the Web that we envision. Problems lie in several other areas such as modeling the information sources, resolving semantic heterogeneity amongst different sources, query planning to gather the requested information from different sites, and intelligently caching retrieved data, to name a few. The focus of this paper is solely on wrapper generation.

The rest of this paper is organized as follows. Section 2 provides an overview of the different kinds of information sources on the Web. Section 3 describes how we semi-automatically generate wrappers. Section 4 presents experimental results to demonstrate the effectiveness of our techniques for wrapper generation. Section 5 describes related work. Section 6 presents future directions and conclusions.

2. Types of Web Information Sources

We categorize the types of pages from Web sources into three classes: multiple-instance sources, single-instance sources, and loosely-structured sources. Certain sources provide information in multiple pages, all conforming to the same format. We call such sources *multiple-instance* sources. Consider a source such as the *CIA World Fact Book*. This source provides information on each of the 267 countries in the world, with information for each country presented on a separate page for that country. The information on each page is presented in a semi-structured manner since each page can be clearly sub-divided into

distinct sections with headings labeling the beginning of each section. Also, the information on all pages is presented in *exactly* the same format. A page for one country is shown in Figure 2. There are clearly identifiable sections such as Geography, Area, Land boundaries, etc., on each page. For each individual page we would like the wrapper to handle queries about one or more sections in the page. For example, "Find the *Land boundaries* and *Area of France*." This wrapper will in turn allow a mediator to handle aggregate queries (spanning multiple countries) such as "Find the *National Product*, and *Defense Expenditures* of all countries in Europe."

There are a number of sources on the Web that fall in the multiple instance category, such as the National Science Foundation (NSF) Grants database,² the General Services Administration (GSA) On-line Shopping database,³ the NSF Funding Opportunities database, Genetics databases such as OMIM,⁴ or the Air Force Fact Sheets,⁵ to name a few. It might be argued that for this category of sources, the information that is put on-line often comes from a database itself. Thus we should query the databases directly. Unfortunately for most of these sources, access to the underlying databases is simply not permitted or might be allowed only with a license fee to query the database. However the information put on-line is readily and freely accessible, which makes a case for building wrappers in order to query these sources.

Another category is that of semi-structured *single instance* pages. There are numerous sources on the Web that contain useful information in a semi-structured form, but on a single page. To name a few, consider the CoopIS 96 proceedings page, list of AAAI Fellows or the Yahoo list of countries by region. The CoopIS '96 proceedings page⁶ is organized into clearly identifiable sections, with a heading for each section (such as Classification and Ontologies, or Data Integration etc.). Each section starts with the Chair of that section followed by papers presented in that section. From such a page we would like a wrapper to be able to answer queries such as "Find the names of all people who chaired a session in CoopIS 96" and expect the wrapper to extract and return the list of chairs i.e., "Witold Litwin, James Geller, Klemens Bohm ...".

Finally there are pages that are more loosely structured, such as a personal homepage. For such cases, i.e., in the absence of clearly identifiable sections with headings, the extraction task becomes much harder. Also the use of fancy

² <http://cos.gdb.org/best/fcdfund/nsf-intro.html>

³ <http://www.fss.gsa.gov/>

⁴ <http://www3.ncbi.nlm.nih.gov/Omim>

⁵ http://www.af.mil/pa/indexpages/fs_index.html

⁶ <http://sunsite.informatik.rwth-aachen.de/dblp/db/conf/coopis/coopis96.html#ScheuermannLC96>

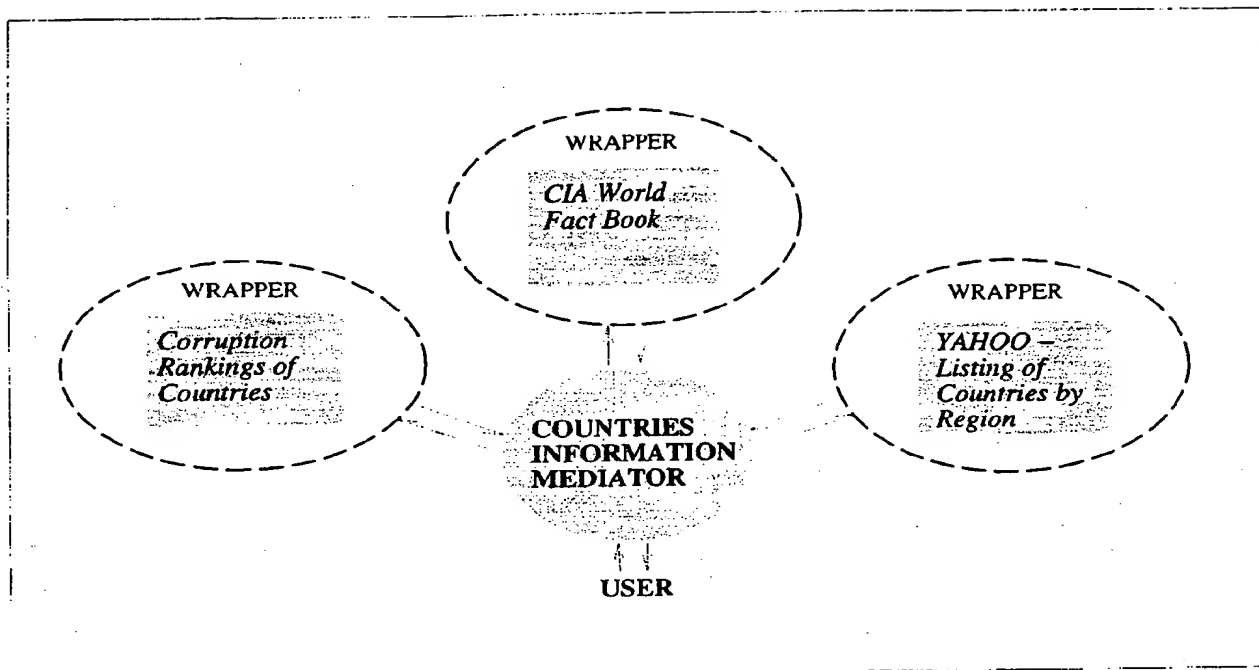


Figure 1. Role of wrappers in providing integrated access to multiple information sources

France

Geography

Location: Western Europe, bordering the Bay of Biscay and English Channel, between Belgium and Spain southeast of the UK; bordering the Mediterranean Sea, between Italy and Spain

Map references: Europe

Area:

total area: 547,030 sq km

land area: 545,630 sq km

comparative area: slightly more than twice the size of Colorado

note: includes Corsica and the rest of metropolitan France, but excludes the overseas administrative divisions

Land boundaries: total 2,892.4 km, Andorra 60 km, Belgium 620 km, Germany 451 km, Italy 488 km, Luxembourg 73 km, Monaco 4.4 km, Spain 623 km, Switzerland 573 km

Coastline: 3,427 km (mainland 2,783 km, Corsica 644 km)

Figure 2. Snapshot of a page from the *CIA World Fact Book*

graphics or images for information presentation makes the task of building a wrapper for that source more difficult.

In this paper we focus on semi-automatically building wrappers for semi-structured sources, in both the multiple-instance and single-instance categories. For loosely structured sources or sources with complicated graphics we have to build wrappers manually. However it is the large number of sources in the semi-structured category, and the wealth of information that can be obtained from them that has motivated us to automate the task of wrapper generation for such sources.

3. Approach to Automated Wrapper Generation

This section describes our approach to generating wrappers for Web sources. We have attempted to automate the process of building wrappers as much as possible. The following steps are involved in generating a wrapper for a new Web source:

- *Structuring the source:* This involves identifying sections and sub-sections of interest on a page.
- *Building a parser for the source pages:* After structuring the source we build a parser that can extract selected sections from a page from the source.
- *Adding communication capabilities between the web sources, wrappers, and mediators:* The mediator that integrates several sources must be able to communicate with the wrappers for these sources. Also, the wrappers must communicate with Web sources to retrieve data in order to answer queries.

We describe these steps in detail below.

3.1. Structuring the Source

In specifying the structure of a page on the Web, two things need to be clearly identified:

1. *Tokens of interest on a page.* By tokens we mean words or phrases that indicate the heading of a section, such as Geography, Economy, or Total Area on the *CIA World Fact Book* page. A heading indicates the beginning of a new section; thus identifying headings identifies the sections on a page.
2. *The nesting hierarchy within sections.* Once a page has been decomposed into various sections, we have to identify the nesting structure of the sections. For instance a *CIA World Fact Book* page is comprised of the sections Geography, People, Economy, Government and Transportation.

The Geography section in turn is broken down into the sections Area, Land boundaries, etc., while Area contains land area, total area, etc.

The structuring task can be done automatically or with minimal user interaction. The key idea here is that a program analyses the HTML and other formatting information in a sample page from the source and guesses the interesting tokens on that page. The system also uses the formatting information to guess the nesting structure of the page. The heuristics used for identifying important tokens on a page and the algorithm used to organize sections into a nested hierarchy are an important contribution of this work. We describe them in more detail below.

3.1.1 Identifying Tokens

Tokens identifying the beginning of a section are often presented in bold font in HTML. They may also be written entirely in upper case words, or may end with a colon. We can generate a lexical analyzer that searches a page for such tokens using LEX [14], a lexical analyzer generator. In Table 1 we list the regular expressions given as specifications to LEX to identify tokens indicating headings on a page. From these specifications we generate a lexical analyzer that identifies words or phrases conforming to the regular expressions. When structuring any page, the system is able to identify headings that are formatted in any of the ways listed in Table 1. For instance, given a page from the *CIA World Fact Book* the system is able to identify the tokens of interest such as Geography, Land boundaries, Area etc. Since each token marks the beginning of a section on a page, at the end of the above tokenizing step all the different sections on a page have been identified.

3.1.2 Determining the Hierarchical Structure

The next step is to obtain the nesting hierarchy of sections on the page, i.e., what sections comprise the page at the top level, what sub-sections comprise other sections in the page, etc. As with the tokenizing step, the nesting hierarchy can be obtained in a semi-automatic fashion for a large number of pages. We have developed an algorithm that, given a page with all sections and headings correctly identified, outputs a hierarchy of sections. The following two simple heuristics are used:

1. *Font Size* - The font of the heading of a sub-section is generally smaller than that of its parent section.
2. *Indentation* - Indentation spaces (which can be detected from raw text or HTML tags) are often used to indicate that one section is a subsection of another.

Regular Expression given as LEX Specification	Description	Example Heading
'<' [bB] [^<]* '>' [^\\n]+ '<' '/' [bB] '>'	Headings in bold tags	Chair
'<' [hH] [0-6] [^<]* '>' [^\\n]+ '<' '/' [hH] [0-6] '>'	Headings with font size	<h3>Geography</h3>
'' [^\\n]+ ''	Headings in Strong Tags	Area
'' [^\\n]+ ''	Strong tags in different case	Population
'' [^\\n]+ ''	Strong tags in lower case	Deadlines
[A-Za-z0-9\\- _]+[:]	Words ending in colon	IRS NUMBER:
'<' [iI] [^<]* '>' [^\\n]+ '<' '/' [iI] '>'	Italicized words	<i>total area:</i>

Table 1. Heuristics for identifying tokens when structuring a page

```

current_node= make_new_tree(); /* returns a node that is the root of a new tree */
while(more_headings){

    new_node=construct_node(heading); /* makes a new node for the new section */
    while ((size_of(current_node) <= size_of(new_node)) or
        (indentation_of(current_node) >= indentation_of(new_node))){
        /* search for the immediate parent section of the new section */
        current_node=parent_of(current_node);
    }
    make_rightmost_child(current_node,new_node);
    /* make the new section the rightmost child of its immediate parent */
    current_node=new_node;
}

generate_grammar();
/* a procedure that from the tree constructed above, for each node N with ordered children
C1, C2 ... , Cm outputs a grammar rule of the form N --> C1 C2 ..... Cm */

```

Figure 3. Algorithm to obtain nesting hierarchy

```

CIApage -> Geography People Government Economy Transportation

Geography -> Location Map_references Area Land_boundaries Coastline ..

Area -> total_area land_area comparative_area
...

```

Figure 4. Nesting Hierarchy for CIA Page

Using the procedure shown in Figure 3, the system outputs a grammar describing the nesting hierarchy of sections in a page. This procedure first builds a tree that reflects the nesting hierarchy of sections. We construct a node for each heading that identifies a new section, and make this node a child of the section that should be its immediate parent based on the font size and indentation of the section headings. The children of each node are ordered, i.e., they appear in the same order in which the corresponding sections appear on the page. When all nodes for all sections have been placed in the tree, the procedure outputs grammar rules for each node in the tree, essentially stating that the section at each node has as sub-sections all its immediate children in the tree (and in the order in which they appear in the tree). For instance, for pages from the *CIA World Fact Book* the grammar output is shown in Figure 4.

It is possible for the system to make mistakes when trying to identify the structure of a new page. Based on the heuristics listed in Table 1, the system can identify headings erroneously (that is identify some words or phrases as headings when they are not, or fail to identify phrases that are headings, but do not conform to any of the regular expressions in Table 1). We have provided a facility for the user to interactively correct the system's guesses. Through a graphical interface the user can highlight tokens that the system misses, or delete tokens that the system erroneously chooses. The user can similarly correct errors in the system-generated grammar that describes the structure of the page.

3.2. Building a Parser for the Source Pages

The next step is to generate a parser for pages from the source. Given a page from the source, such a parser can extract any selected section(s) from the page. For instance a parser for pages from the *CIA World Fact Book* can extract sections such as Geography, Area (the "." indicates that Area is a subsection of Geography in the spirit of complex objects) i.e., the Area sub-section within the Geography section from the page for any country. Such a parser can be automatically generated, since all of the grammatical and lexical information needed to parse the page is obtained at the structuring step. The compiler generator YACC [10] and the tool LEX are used for this purpose. The tokens identified in the structuring step are directly input as specifications to LEX to generate a lexical analyzer for a page from the source. For instance the tokens identified in the *CIA World Fact Book* page are Geography, Location, Map references, Area, total area, etc., and the specifications given to LEX to generate a lexical analyzer for a page from the *CIA World Fact Book* are shown in Figure 5.

```
<h3>Geography</h3>{return(GEO_HEAD);}
<b>Location:</b>{return(LOC_HEAD);}
<b>Map references:</b>{return(MAP_HEAD);}
<b>Area:</b>{return(AREA_HEAD);}
<i>total area:</i>{return(TOT_HEAD);}
...
. {return(TEXT);}
\n {return(TEXT);}
```

Figure 5. LEX Specifications for CIA Page

The tool YACC can generate a parser for a language given grammar rules that specify valid sentences in the language. We directly translate the grammar rules describing the overall structure of the page, obtained at the end of the structuring step, into a YACC specification. The parser generated can parse valid "sentences" i.e., pages from the source. Figure 6 shows what the rules specified to YACC to parse pages from the *CIA World Fact Book* look like. For instance, the first part of the first rule states that a single page is comprised of the Geography section, People section, etc. The second part of the rule shows YACC code for storing and manipulating parsed data. With these specifications we use LEX and YACC to generate a parser for pages from the source.

3.3. Adding Communication Capabilities between the Wrapper, Mediator and Web Sources

Given a query, a wrapper for a Web source should be able to fetch the pages containing the requested information from the Web source. Also some mechanism is needed for communication between the mediator and the wrapper as they are separate processes, possibly running at different locations. The following communication functionality thus needs to be added to the wrapper.

1. *Identifying network locations of page(s) needed to answer a query.* For sources with just a single page this is straightforward i.e., the URL for that page is known to the wrapper. For sources with multiple pages, a mapping between a query and the URL of the relevant page might be required. For instance for the *CIA World Fact Book* there is a one to one mapping between the country name and the URL of the page for that country. This mapping can be obtained from the index page for the CIA source. For the GSA database the *part number* appears at the end of the URL for that source to point to the page for that part.

To provide the capability of determining the network location of the page relevant to a query, the user spec-

CIApage	:Geographysection Peoplesection Governmentsection Economysection Transportsection {strcpy(\$\$, \$1); strcat(\$\$, \$2); strcat(\$\$, \$3); strcat(\$\$, \$4); }
Geographysection	: Locationsection Maprefsection Areasection Landboundariessection... {strcpy(\$\$, \$1); ... }
Areasection	:totalareasection landareasection compareasection {strcpy(\$\$, \$1); strcat(\$\$, \$2); strcat(\$\$, \$3); }
Locationsection	: Locationheading Text {strcpy(\$\$, \$1); strcat(\$\$, \$2); }
Maprefsection	: Maprefheading Text {strcpy(\$\$, \$1); strcat(\$\$, \$2); }
Locationheading	: LOC_HEAD { strcpy(\$\$, yytext); }
Maprefheading	: MAP_HEAD { strcpy(\$\$, yytext); }
...	

Figure 6. YACC specifications for CIA page

ifies a *mapping function* which takes necessary arguments from a query (eg. *country name* from a query on the CIA source) and constructs a URL pointing to the page to be fetched.

2. *Capability to retrieve data over the network.* Currently we are using PERL scripts for the purpose of making HTTP connections to the Web information sources and retrieving data from them.
3. *Communication between the mediator and wrapper.* We are using the agent communication language KQML [8] for the purpose of providing interprocess communication between the mediator and a wrapper.

Adding the above functionality is the final step in generating a wrapper for a new source. The parser for pages from a Web source plus the above communication functionality results in a complete wrapper for that Web source.

4. Results

We have applied the wrapper generator to the task of generating wrappers for a variety of internet sources. We present experimental results to provide an idea of the effort required to generate a wrapper for a new source. The step that is most difficult to automate when generating a wrapper is the first step where we obtain the structure of a page or sample pages from the source. Generating the parser is then done automatically and defining a mapping function from

queries to URLs of relevant for sources with multiple pages requires comparatively little effort on part of the user. It is thus the structuring step that dominates the time and effort needed to build a wrapper for a new source.

We used the wrapper generator to build wrappers for several internet sources and to evaluate the effectiveness of the heuristics we use for structuring a new page automatically. To provide a quantitative measure of the effectiveness of the heuristics, we define what we call *correction steps*. During structuring a page, each time the user has to manually correct a token (i.e., add or delete a token) or correct a rule in the grammar describing the nesting hierarchy of sections, it is counted as one correction step. The total number of correction steps made before the page is completely structured provides an estimate of how hard it is to automatically structure that page. We also provide the time taken to generate the wrapper for each source. This would of course vary from user to user. Nevertheless, the results give a sense for approximately how long it might take to generate wrappers using this toolkit.

Table 2 demonstrates the ease with which we built sources for a dozen internet sites, from both the multiple-instance and single-instance categories. We provide the number of correction steps to structure a sample page from each source as well as the total time (in minutes) taken to build a wrapper for that source. The results are extremely encouraging. Several sources require almost no or very few correction steps to structure them, thus showing that the heuristics for structuring pages are quite successful. Also, it takes only a few minutes to generate a wrapper for most

<i>Multiple-instance sources</i>	<i>Correc- tion steps</i>	<i>Time in min</i>	<i>Single instance sources</i>	<i>Correc- tion steps</i>	<i>Time in min</i>
1. The CIA World Fact Book.	0	2	1. CoopIS 96 Proceedings page	1	3
2. GSA On-line Shopping database.	2	5	2. AAAI-97 conference homepage	3	3
3. The NSF database.	0	5	3. List of US Universities by state	6	4
4. The OMIM Genetics database	4	4	4. List of AAAI Fellows by year	0	1
5. Hoover Company Profiles	2	4	5. Computer Science Job Listings	6	5
6. The Internet Movie Database	3	6	6. SIGMOD Record page	4	3
7. The Air Force Library Fact Sheets	0	2	7. US Air Force Organization page	0	1

Table 2. Experimental Results showing the effort and time to build wrappers for different sources

new sources. Such a toolkit is thus extremely useful as it provides a convenient and quick way to generate wrappers for new sources of information on the Web and then integrate them via a mediator. We successfully integrated several sources in the countries information domain, such as the *CIA World Fact Book*, Yahoo listings of countries by region etc. using the Ariadne system, which is a descendant of the SIMS [3] information mediator that addresses the problem of integrating Web sources. We were then able to pose queries to Ariadne such as "Find the *External debt* and *Defense expenditures* of all countries in the EEC." The answer given by the mediator is shown in Figure 7.

5. Related work

Generating wrappers for databases and Web sources, and providing database like querying for semi-structured data are research areas that have received considerable attention recently. Hammer et al. [9] developed a *template-based* approach to generating wrappers for Web sources and other types of legacy systems. With their approach, the user provides actions for the system to execute when a query matches a certain template or format. This approach provides a way of rapidly constructing wrappers by example, but it could require a large number of examples to specify a single source.

Doorenbos et al. [6] developed an Internet comparison shopping agent that can automatically build wrappers for Web sites. Since they focus on pages that contain items for sale, they make much stronger assumptions about the type

of information they are looking for and use that information to hypothesize the underlying structure. Their wrapper language is not very expressive and the system is quite limited in terms of the types of pages for which it can generate wrappers.

Kushmerick et al. [13] also developed an approach to automatically generating wrappers. The focus of their work is very similar to ours i.e., building wrappers for Web sources to be integrated by a software agent. However, they follow a very different approach that uses inductive learning techniques to build a program that extracts data from a Web page. They assume that they are given a set of recognizers that can then be used to generate examples for the learning system. The advantage of their approach is that the resulting wrappers will be more robust to inconsistencies across multiple-instance pages. On the other hand, their approach could not be used to generate wrappers for more complex pages, such as the *CIA World Fact Book*, without first building recognizers for each of the fields of those pages.

There is also a variety of work that addresses issues in directly querying semi-structured data, particularly data obtained from Web sources, in a database-like fashion [1, 5, 2, 12, 16]. These efforts are concerned with issues such as the development of data models and query languages for semi-structured data, defining formal semantics for these query languages, and efficiently implementing these languages. The focus of our work is on the generation of wrappers that provide a uniform interface to a variety of semi-structured data, as opposed to efforts that support direct querying of the data.

Country	EXTERNAL DEBT	DEFENSE EXPENDITURES
AUSTRIA	\$21.5 billion (1994 est.)	exchange rate conversion - about \$1.8 billion, 0.9% of GDP (1994)
BELGIUM	\$31.3 billion (1992 est.)	exchange rate conversion - \$3.9 billion, 1.8% of GDP (1994)
DENMARK	\$40.9 billion (1994 est.)	exchange rate conversion - \$2.7 billion, 1.9% of GDP (1994)
FINLAND	\$30 billion (December 1993)	exchange rate conversion - \$1.86 billion, about 1.9% of GDP (1994)
FRANCE	\$300 billion (1993 est.)	exchange rate conversion - \$47.1 billion, 3.1% of GDP (1995)
GERMANY	\$NA	exchange rate conversion - \$40 billion, 1.8% of GNP (1995)
GREECE	\$26.9 billion (1993)	exchange rate conversion - \$4.1 billion, 5.4% of GDP (1994)
IRELAND	\$20 billion (1994 est.)	exchange rate conversion - \$500 million, 1.3% of GDP (1994)

Figure 7. Answer to query involving multiple Web sources

6. Future work and conclusions

We have presented the ideas and results of our approach for automatically generating wrappers for Web sources. We have clearly separated the tasks in building wrappers that are specific to a particular Web source such as structuring the source, and tasks which are repetitive for any source (and can thus be done by the system) such as generating a parser from the structure of a page and adding communication capabilities. The main contribution of our work is automating the structuring step, through the use of heuristics for determining the structure by exploiting formatting information in pages from the source. Our ideas appear to be effective for many types of semi-structured sources. However we need more advanced wrappers to be able to broaden the scope of sources we can generate wrappers for and also to be able to handle finer grained queries. Currently we are working on enhancing the wrappers with the following capabilities:

- *Learning new tokens by examples:* It is possible that while structuring a page, the system is unable to identify tokens on the page if they do not conform to any of the regular expressions in Table 1. We are working on adding capabilities to the system to quickly *learn* the structure of a new kind of heading from a few user examples. We are applying techniques for inducing Hidden Markov models (HMMs), describing the tokens, from corpora of positive examples. The basic idea, described in [17] is to start with an HMM accepting only the initial tokens marked by the user. Then, states in the HMM are *merged* to yield a generalized model that can be used to identify the remaining tokens in the page. The system can then identify the remaining tokens in the page automatically.
- *Handling Tables:* A challenging problem is to automatically build parsers for information in tables. The hard problem here is to determine exactly what is contained in the different rows and columns of the table and then build a parser to extract information from it.

- *Handling finer grained queries:* Consider the Land boundaries section on a CIA World Fact Book page. Currently the wrapper cannot handle queries such as 'Find the names of all countries bordering France' as the parser does not have enough knowledge of the structure *within* the Land boundaries field to decompose that field into pairs of countries and corresponding border lengths. We are currently investigating using machine learning techniques where the user gives a few examples highlighting items of interest within a field and the system is eventually able to learn the structure within that field.

We are using the wrapper generator system to generate wrappers for semi-structured sources and are working on making the system more advanced and capable of handling more kinds of Web sources. Generation of wrappers is very useful in meeting our broader goal of integrating Web sources via a mediator, by which we hope to simplify the task of obtaining information from the already numerous and ever growing information sources on the Web.

Acknowledgments

We would like to thank Steve Minton and other members of the SIMS and Ariadne projects for their helpful contributions to this work. We also wish to thank Vipul Kashyap of the InfoSleuth project at MCC for suggestions on future enhancements.

References

- [1] S. Abiteboul. Querying semi-structured data. In *ICDT (invited talk)*, 1997.
- [2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Weiner. The Lorel query language for semistructured data. *Journal on Digital Libraries*, To appear.

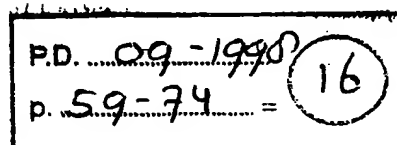
- [3] Y. Arens, C. A. Knoblock, and W.-M. Shen. Query reformulation for dynamic information integration. *Journal of Intelligent Information Systems, Special Issue on Intelligent Information Integration*, 6(2/3):99–130, 1996.
- [4] R. Bayardo, W. Bohrer, R. Brice, A. Cichocki, G. Fowler, A. Helal, V. Kashyap, T. Ksiczky, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk. Semantic integration of information in open and dynamic environments. Technical Report MCC-INSL-088-96, MCC, Austin, Texas, 1996.
- [5] P. Buneman, S. Davidson, and G. H. D. Suci. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, 1996.
- [6] R. B. Doorenbos, O. Etzioni, and D. S. Weld. A scalable comparison-shopping agent for the world-wide web. In *Proceedings of the First International Conference on Autonomous Agents*, Marina del Rey, CA, 1997.
- [7] O. Etzioni and D. S. Weld. A softbot-based interface to the Internet. *Communications of the ACM*, 37(7), 1994.
- [8] T. Finin, Y. Labrou, and J. Mayfield. KQML as an agent communication language. In J. Bradshaw, editor, *Software Agents*. AAAI/MIT Press, Menlo Park, CA, in press.
- [9] J. Hammer, M. Brenning, H. Garcia-Molina, S. Nesterov, V. Vassalos, and R. Yemeni. Template-based wrappers in the tsimmi system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (Demonstration Track)*, Tucson, AZ, 1997.
- [10] S. C. Johnson. Yacc: Yet another compiler compiler. Technical Report CSTR 32, AT&T Bell Laboratories, 1978.
- [11] T. Kirk, A. Y. Levy, Y. Sagiv, and D. Srivastava. The information manifold. In *Working Notes of the AAAI Spring Symposium on Information Gathering in Heterogeneous, Distributed Environments*, Technical Report SS-95-08, AAAI Press, Menlo Park, CA, 1995.
- [12] D. Konopnicki and O. Shemueli. W3QS: A query system for the World Wide Web. In *Proceedings of the 21st International Conference on Very Large Databases*, Zurich, Switzerland, 1995.
- [13] N. Kushmerick, D. S. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *International Joint Conference on Artificial Intelligence (IJCAI)*, Nagoya, Japan, 1997.
- [14] M. E. Lesk. Lex - a lexical analyzer generator. Technical Report CSTR 39, AT&T Bell Laboratories, Murray Hill, New Jersey, 1975.
- [15] E. Mena, V. Kashyap, A. Sheth, and A. Illarramendi. Observer: an approach for query processing in global information systems based on interoperation across pre-existing ontologies. In *Proceedings of the First IFCIS International Conference on Cooperative Information Systems (CoopIS '96)*, June, 1996.
- [16] A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the world wide web. In *Symposium on Parallel and Distributed Information Systems*, Miami, Florida, 1996.
- [17] A. Stolcke and S. Omohundro. Inducing probabilistic grammars by bayesian model merging. In R.C. Carrasco and J. Oncina, editors, *Grammatical Inference and Applications*, pages 106–118. Springer, 1994.
- [18] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, March 1992.

Database Techniques for the World-Wide Web: A Survey

Daniela Florescu
Inria Roquencourt
dana@rodin.inria.fr

Alon Levy
Univ. of Washington
alon@cs.washington.edu

Alberto Mendelzon
Univ. of Toronto
mendel@cs.toronto.edu



1 Introduction

The popularity of the World-Wide Web (WWW) has made it a prime vehicle for disseminating information. The relevance of database concepts to the problems of managing and querying this information has led to a significant body of recent research addressing these problems. Even though the underlying challenge is the one that has been traditionally addressed by the database community – how to manage large volumes of data – the novel context of the WWW forces us to significantly extend previous techniques. The primary goal of this survey is to classify the different tasks to which database concepts have been applied, and to emphasize the technical innovations that were required to do so.

We do not claim that database technology is the magic bullet that will solve all web information management problems; other technologies, such as Information Retrieval, Artificial Intelligence, and Hypertext/Hypermedia, are likely to be just as important. However, surveying all the work going on in these areas, and the interactions between them and database ideas, would be far beyond our scope.

We focus on three classes of tasks related to information management on the WWW.

Modeling and querying the web: Suppose we view the web as a directed graph whose nodes are web pages and whose edges are the links between pages. A first task we consider is that of formulating queries for retrieving certain pages on the web. The queries can be based on the *content* of the desired pages and on the *link structure* connecting the pages. The simplest instance of this task, which is provided by search engines on the web is to locate pages based on the words they contain. A simple generalization of such a query is to apply more complex predicates on the contents of a page (e.g., find the pages that contain the word "Clinton" next to a link to an image). Finally, as an example of a query that involves the structure of the pages, consider the query asking for all images reachable from the root of the CNN web site within 5 links. The last type of queries are especially useful when detecting violations of integrity constraints on a web site or a collection of web sites.

Information extraction and integration: Certain web sites can be viewed at a finer granularity level than pages, as *containers* of structured data (e.g., sets of tuples, or sets of objects). For example, the Internet Movie Database (<http://www.imdb.com>) can be viewed as a front end interface to a database about movies. Given the rise in the number of such sites, there are two tasks we consider. The first task is to actually extract a structured representation of the data (e.g., a set of tuples) from the HTML pages containing them. This task is performed by a set of *wrapper* programs, whose creation and maintenance raises several challenges. Once we view these sites as autonomous heterogeneous databases, we can address the second task of posing queries that require the integration of data. The second task is addressed by *mediator* (or *data integration*) systems.

Web site construction and restructuring: A different aspect in which database concepts and technology can be applied is that of building, restructuring and managing web-sites. In contrast to the previous two classes which apply on *existing* web sites, here we consider the process of *creating* sites. Web sites can be constructed either by starting with some raw data (stored in databases or structured files) or by restructuring existing web sites. Performing this task requires methods for *modeling* the structure of web site and languages for *restructuring* data to conform to a desired structure.

Before we begin, we note that there are several topics concerning the application of database concepts to the WWW which are not covered in this survey, such as caching and replication (see [WWW98, GRC97] for recent works), transaction processing and security in web environments (see e.g. [Bil98]), performance, availability and scalability issues for web servers (e.g. [CS98]), or indexing techniques and crawler technology (e.g. [CGMP98]). (Furthermore, this is not meant to be a survey on existing products even in the areas on which we do focus. Finally, there are several tangential areas whose results are applicable to the systems we discuss, but we do not cover them here. Examples of such fields include systems for managing document collections and ranking of documents (e.g., Harvest [BDH⁺95], Gloss [GGMT99]) and flexible query answering systems [BT98]. Finally, the field of web/db is a very dynamic one; hence, there are undoubtedly some omissions in our coverage, for which we apologize in advance.

The survey is organized as follows. We begin in Section 2 by discussing the main issues that arise in designing data models for web/db applications. The following three sections

consider each of the aforementioned tasks. Section 6 concludes with perspectives and directions for future research.

2 Data Representation for Web/DB Tasks

Building systems for solving any of the previous tasks requires that we choose a method for modeling the underlying domain. In particular, in these tasks, we need to model the web itself, structure of web sites, internal structure of web pages, and finally, contents of web sites in finer granularities. In this section we discuss the main distinguishing factors of the data models used in web applications.

Graph data models: As noted above, several of the applications we discuss require to model the set of web pages and the links between them. These pages can either be on several sites or within a single site. Hence, a natural way to model this data is based on a *labeled graph* data model. Specifically, in this model, nodes represent web pages (or internal components of web pages), and arcs represent links between pages. The labels on the arcs can be viewed as attribute names. Along with the labeled graph model, several query languages have been developed. One central feature that is common to these query languages is the ability to formulate *regular path expression* queries over the graph. Regular path expressions enable posing navigational queries over the graph structure.

Semistructured data models: The second aspect of modeling data for web applications is that in many cases the structure of the data is irregular. Specifically, when modeling the structure of a web site, we don't have a fixed schema which is given in advance. When modeling data coming from multiple sources, the representation of some attributes (e.g., addresses) may differ from source to source. Hence, several projects have considered models of *semistructured data*. The initial motivation for this work was the existence and relative success of permissive data models such as [TMD92] in the scientific community, the need for exchanging objects across heterogeneous sources [PGMW95], and the task of managing document collections [MP96].

Broadly speaking, semistructured data refers to data with some of the following characteristics:

- the schema is not given in advance and may be implicit in the data,
- the schema is relatively large (w.r.t. the size of the data) and may be changing frequently,
- the schema is *descriptive* rather than *prescriptive*, i.e., it describes the current state of the data, but violations of the schema are still tolerated,
- the data is not strongly typed, i.e., for different objects, the values of the same attribute may be of differing types.

Models for semistructured data have been based on labeled directed graphs [Abi97, Bun97].¹ In a semistructured data model, there is no restriction on the set of arcs that emanate from a given node in a graph, or on the types of the values of

¹It should be noted that there is no inherent difficulty in translating these models into relational or object-oriented terms. In fact, the languages underlying Description Logics (e.g., Classic [BBMR89]) and FLORID [HLLS97] have some of the features mentioned above, and are described in non-graph models.

attributes. Because of the characteristics of semistructured data mentioned above, an additional feature that becomes important in this context is the ability to query the schema (i.e., the labels on the arcs in the graph). This feature is supported in languages for querying semistructured data by *arc variables* which get bound to labels on arcs, rather than nodes in the graph.

In addition to developing models and query languages for semistructured data, there has been considerable recent work on issues concerning the management of semistructured data, such as the extraction of structure from semistructured data [NAM98], view maintenance [ZGM98, AMR⁺98], summarization of semistructured data ([BDFS97, GW97]), and reasoning about semistructured data [CGL98, FFLS98]. Aside from the relevance of these works to the tasks mentioned in this survey, the systems based on these methods will be of special importance for the task of managing large volumes of XML data [XML98].

Other characteristics of web data models: Another distinguishing characteristic of models used in web/db applications is the presence of web-specific constructs in the data representation. For example, some models distinguish a unary relation identifying pages and a binary relation for links between pages. Furthermore, we may distinguish between links within a web site and external links. An important reason to distinguish a link relation is that it can generally only be traversed in the forward direction. Additional second order dimensions along which the data models we discuss differ are (1) the ability to model order among elements in the database, (2) modeling nested data structures, and (3) support for collection types (sets, bags, arrays). An example of a data model that incorporates explicit web-specific constructs (pages and page schemes), nesting, and collection types is ADM, the data model of the ARANEUS project [AMM97b]. We remark that all the models we mention in this paper represent only static structures. For example, the work on modeling the structure of web sites do not consider dynamic web pages created as a result of user inputs.

An important aspect of languages for querying data in web applications is the need to create complex structures as a result of a query. For example, the result of a query in a web site management system is the graph modeling the web site. Hence, a fundamental characteristic of many of the languages we discuss in this paper is that their query expressions contain a *structuring* component in addition to the traditional data filtering component.

Table 1 summarizes some of the web query systems covered in this paper. A more detailed version of this table, <http://www.cs.washington.edu/homes/alon/webdb.html> includes URLs for the systems where available. In subsequent sections we will illustrate in detail languages for querying data represented in these models.

3 Modeling and Querying the Web

If the web is viewed as a large, graph-like database, it is natural to pose queries that go beyond the basic information retrieval paradigm supported by today's search engines and take structure into account; both the internal structure of web pages and the external structure of links that interconnect them. In an often-cited paper on the limitations of hypertext systems, Halasz says: [Hal88]

System	Data Model	Language Style	Path Expressions	Graph Creation
WebSQL [MMM97]	relational	SQL	Yes	No
W3QS [KS95]	labeled multigraphs	SQL	Yes	No
WebLog [LSS96]	relational	Datalog	No	No
Lorel [AQM ⁺ 97]	labeled graphs	OQL	Yes	No
WebOQL [AM98]	hypertrees	OQL	Yes	Yes
UnQL [BDHS96]	labeled graphs	structural recursion	Yes	Yes
STRUDEL [FFK ⁺ 98, FFLS97]	labeled graphs	Datalog	Yes	Yes
ARANEUS (ULIXES) [AMM97b]	page schemes	SQL	Yes	Yes
FLORID [HLLS97]	F-logic	Datalog	Yes	No

Table 1: Comparison of query systems

Content search ignores the structure of a hypermedia network. In contrast, structure search specifically examines the hypermedia structure for subnetworks that match a given pattern.

and goes on to give examples where such queries are useful.

3.1 Structural Information Retrieval

The first tools developed for querying the web were the well-known search engines which are now widely deployed and used. These are based on searching indices of words and phrases appearing in documents discovered by web "crawlers." More recently, there have been efforts to overcome the limitations of this paradigm by exploiting link structure in queries. For example, [Kle98], [BH98] and [CDRR98], propose to use the web structure to analyze the many sites returned by a search engine as relevant to a topic in order to extract those that are likely to be authoritative sources on the topic. To support connectivity analysis for this and other applications, (such as efficient implementations of the query languages described below) the Connectivity Server [BBH⁺98] provides fast access to structural information. Google [BP98], a prototype next-generation web search engine, makes heavy use of web structure to improve crawling and indexing performance. Other methods for exploiting link structure are presented in [PPR96, CK98]. In these works, structural information is mostly used behind the scenes, to improve the answers to purely content-oriented queries. Spertus [Spe97] points out many useful applications of queries that take link structure into account explicitly.

3.2 Related query paradigms

In this section we briefly describe several families of query languages that were not developed specifically for querying the web. However, since the concepts on which they are based are similar in spirit to the web query languages we discuss, these languages can also be useful for web applications.

Hypertext/document query languages: A number of models and languages for querying structured documents and hypertexts were proposed in the pre-web era. For example, Abiteboul et al. [ACM93] and Christophides et al. [CACS94] map documents to object oriented database instances by means of semantic actions attached to a grammar. Then the database representation can be queried using the query language of the database. A novel aspect of this approach is the possibility of querying the structure by means of path vari-

ables. Guting et al. [GZC89] model documents using nested ordered relations and use a generalization of nested relational algebra as a query language. Beeri and Kornatzky [BK90] propose a logic whose formulas specify patterns over the hypertext graph.

Graph query languages: Work in using graphs to model databases, motivated by applications such as software engineering and computer network management, led to the G, G+ and GraphLog graph-based languages [CMW87, CMW88, CM90]. In particular, G and G+ are based on labeled graphs; they support regular path expressions and graph construction in queries. GraphLog, whose semantics is based on Datalog, was applied to Hypertext queries in [CM89]. Paredaens et al [PdBA⁺92] developed a graph query language for object-oriented databases.

Languages for querying semistructured data: Query languages for semistructured data such as Lorel [AQM⁺97], UnQL [BDHS96] and STRUQL [FFLS97] also use labeled graphs as a flexible data model. In contrast to graph query languages, they emphasize the ability to query the schema of the data, and the ability to accommodate irregularities in the data, such as missing or repeated fields, heterogeneous records. Related work in the OO community [Har94] proposes "schema-shy" models and queries to handle information about software engineering artifacts.

These languages were not developed specifically for the web, and do not distinguish, for example, between graph edges that represent the connection between a document and one of its parts and edges that represent a hyperlink from one web document to another. Their data models, while elegant, are not very rich, lacking such basic comforts as records and ordered collections.

3.3 First generation web query languages

A first generation of web query languages aimed to combine the content-based queries of search engines with structure-based queries similar to what one would find in a database system. These languages, which include W3QL [KS95], WebSQL [MMM97, AMM97a], and WebLog [LSS96], combine conditions on text patterns appearing within documents with graph patterns describing link structure. We use WebSQL as an example of the kinds of queries that can be asked.

WebSQL WebSQL proposes to model the web as a relational database composed of two (virtual) relations: Document and Anchor. The Document relation has one tuple for each document in the web and the Anchor relation has one

tuple for each anchor in each document in the web. This relational abstraction of the web allows us to use a query language similar to SQL to pose the queries.

If Document and Anchor were actual relations, we could simply use SQL to write queries on them. But since the Document and Anchor relations are completely virtual and there is no way to enumerate them, we cannot operate on them directly. The WebSQL semantics depends instead on materializing portions of them by specifying the documents of interest in the FROM clause of a query. The basic way of materializing a portion of the web is by navigating from known URL's. Path regular expressions are used to describe this navigation. An atom of such a regular expression can be of the form $d1 \Rightarrow d2$, meaning document $d1$ points to $d2$ and $d2$ is stored on a different server from $d1$; or $d1 \rightarrow d2$, meaning $d1$ points to $d2$ and $d2$ is stored on the same server as $d1$.

For example, suppose we want to find a list of triples of the form $(d1, d2, label)$, where $d1$ is a document stored on our local site, $d2$ is a document stored somewhere else, and $d1$ points to $d2$ by a link labeled $label$. Assume all our local documents are reachable from `www.mysite.start`.

```
SELECT d.url, e.url, a.label
FROM Document d SUCH THAT
    "www.mysite.start" ->* d,
    Document e SUCH THAT d => e,
    Anchor a SUCH THAT a.base = d.url
WHERE a.href = e.url
```

The FROM clause instantiates two Document variables, d and e , and one Anchor variable a . The variable d is bound in turn to each local document reachable from the starting document, and e is bound to each non-local document reachable directly from d . The anchor variable a is instantiated to each link that originates in document d ; the extra condition that the target of link a be document e is given in the WHERE clause. Another way of materializing part of the Document and Anchor relations is by content conditions: for example, if we were only interested in documents that contains the string "database" we could have added to the FROM clause the condition d MENTIONS "database". The implementation uses search engines to generate candidate documents that satisfy the MENTION conditions.

Other Languages W3QL [KS95] is similar in flavour to WebSQL, with some notable differences: it uses external programs (similar to user defined functions in object-relational languages) for specifying content conditions on files rather than building conditions into the language syntax, and it provides mechanisms for handling forms encountered during navigation. In [KS98], Konopnicki and Shmueli describe planned extensions to move W3QL into what we call the second generation. These include modeling internal document structure, hierarchical web modeling that captures the notion of web site explicitly, and replacing the external program method of specifying conditions with a general extensible method based on the MIME standard.

WebLog [LSS96] differs from the above languages in using deductive rules instead of SQL-like syntax (see the description of FLORID below).

WQL, the query language of the WebDB project [LSCH98], is similar to WebSQL but it supports more comprehensive SQL functionality such as aggregation and grouping, and provides limited support for querying intra-document struc-

ture, placing it closer to the class of languages discussed in the next subsection.

3.4 Second generation: Web Data Manipulation Languages

The languages above treat web pages as atomic objects with two properties: they contain or do not contain certain text patterns, and they point to other objects. Experience with their use suggests there are two main areas of application that they can be useful for: data wrapping, transformation and restructuring, as described in Section 4; and web site construction and restructuring, as described in Section 5. In both application areas, it is often essential to have access to the internal structure of web pages from the query language if we want declarative queries to capture a large part of the task at hand. For example, the task of extracting a set of tuples from the HTML pages of the Internet Movie Database requires parsing the HTML and selectively accessing certain subtrees in the parse tree.

In this section we describe the second-generation of web query languages that we call "Web data manipulation languages." These languages go beyond the first generation languages in two significant ways. First, they provide access to the structure of the web objects that they manipulate. Unlike the first-generation languages, they model internal structure of web documents as well as the external links that connect them. They support references to model hyperlinks, and some support ordered collections and records for more natural data representation. Second, these languages provide the ability to create new complex structures as a result of a query. Since the data on the web is commonly semistructured (or worse), these languages still emphasize the ability to support semistructured features. We briefly describe three languages in this class: WebOQL [AM98], STRUQL [FFLS97] and FLORID [HLLS97].

WebOQL

The main data structure provided by WebOQL is the hypertree. Hypertrees are ordered arc-labeled trees with two types of arcs, internal and external. Internal arcs are used to represent structured objects and external arcs are used to represent references (typically hyperlinks) among objects. Arcs are labeled with records. Figure 1, from [Aro97], shows a hypertree containing descriptions of publications from several research groups. Such a tree could easily be built, for example, from an HTML file, using a generic HTML wrapper.

Sets of related hypertrees are collected into *webs*. Both hypertrees and webs can be manipulated using WebOQL and created as the result of a query.

WebOQL is a functional language, but queries are couched in the familiar select-from-where form. For example, suppose that the name `csPapers` denotes the papers database in Figure 1, and that we want to extract from it the title and URL of the full version of papers authored by "Smith".

```
select [y.Title, y.Url]
from x in csPapers, y in x'
where y.Authors = "Smith"
```

In this query, x iterates over the simple trees of `csPapers` (i.e., over the research groups) and, given a value for x , y

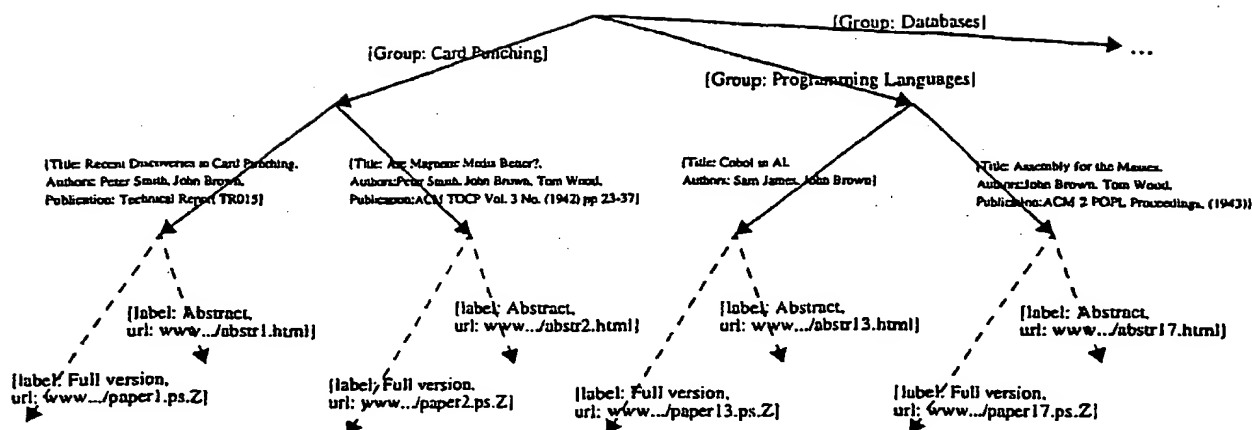


Figure 1: Example of a hypertree

iterates over the simple trees of x' . The primed variable x' denotes the result of applying to tree x the Prime operator, which returns the first subtree of its argument. The same operator is used to extract from tree y its first subtree in y' .Url. The square brackets denote the Hang operator, which builds an arc labeled with a record formed with the arguments (in this example, the field names are inferred.) Finally, the tilde represents the string pattern matching predicate: its left argument is a string and its right argument is a pattern.

Web Creation The query above maps a hypertree into another hypertree; more generally, a query is a function that maps a web into another. For example, the following query creates a new page for each research group (using the group name as URL). Each page contains the publications of the corresponding group.

```
select x' as x.Group
from x in csPapers
```

In general, the select clause has the form 'select q_1 as s_1 , q_2 as s_2 , ..., q_m as s_m ', where the q_i 's are queries and each of the s_i 's is either a string query or the keyword schema. The "as" clauses create the URL's s_1 , s_2 , ..., s_m , which are assigned to the new pages resulting from each query q_i .

Navigation Patterns Navigation patterns are regular expressions over an alphabet of record predicates: they allow us to specify the structure of the paths that must be followed in order to find the instances for variables.

Navigation patterns are mainly useful for two purposes. The first reason is for extracting subtrees from trees whose structure we do not know in detail or whose structure presents irregularities, and the second is for iterating over trees connected by external arcs. In fact, the distinction between internal and external arcs in hypertrees becomes really useful when we use navigation patterns that traverse external arcs. Suppose that we have a software product whose documentation is provided in HTML format and we want to build a full-text index for it. These documents form a complex

hypertext, but it is possible to browse them sequentially by following links having the string "Next" as label. For building the full-text index we need to feed the indexer with the text and the URL of each document. We can obtain this information using the following query:

```
select [ x.Url, x.Text ]
from x in browse("root.html")
via (~*[Text ~ "Next"]>)*
```

StruQL

STRUQL is the query language of the STRUDEL web site management system, described below in Section 5. Even though STRUQL was developed in the context of a specific web application, it is a general purpose query language for semistructured data, based on a data model of labeled directed graphs. In addition, the STRUDEL data model includes named collections, and supports several atomic types that commonly appear in web pages, such as URLs, and Postscript, text, image, and HTML files. The result of a STRUQL query is a graph in the same data model as the input graphs. In STRUDEL, STRUQL was used for two tasks: querying heterogeneous sources to integrate them into a site data graph, and for querying this data graph to produce a site graph.

A STRUQL query is a set of possibly nested blocks, each of the form:

```
[where C1,...,Ck]
[create N1,...,Nn]
[link L1,...,Lp]
[collect G1,...,Gq].
```

The where clause can include either membership conditions or conditions on pairs of nodes expressed using regular path expressions. The where clause produces all bindings of node and arc variables to values in the input graph, and the remaining clauses use Skolem functions to construct a new graph from these bindings.

We illustrate STRUQL with a query defining a web site, starting with a Bibtex bibliography file, modeled as a labeled graph. The web site will consist of three kinds of pages: a PaperPresentation page for each bibliography entry, a Year page for each year, pointing to all papers published in that year, and a Root page pointing to all the Year pages. After showing the query in STRUQL, we show it in WebOQL to give a feel for the differences between the languages.

```
// Create Root
create RootPage()
// Create a presentation for every publication x
where Publications(x), x->1->v
create PaperPresentation(x)
link PaperPresentation(x) -> 1 -> v
{ // Create a page for every year
  where 1 = "year"
  create YearPage(v)
  link
    YearPage(v) -> "Year" -> v
    YearPage(v) -> "Paper" -> PaperPresentation(x),
    // Link root page to each year page
    RootPage() -> "YearPage" -> YearPage(v)
}
```

In the where clause, the notation `Publications(x)` means that `x` belongs to the collection `Publications`, and the atom `x → 1 → v` denotes that there is a link in the graph from `x` to `v` and the label on the arc is `1`. The same notation is used in the link clause to specify the newly created edges in the resulting graph. After creating the Root page, the first CREATE generates a page for each publication (denoted by the Skolem function, `PaperPresentation`). The second CREATE, nested within the outer query, generates a Year page for each year, and links it to the Root page and to the `PaperPresentation` pages of the publications published in that year. Note the Skolem function `YearPage` ensures that a Year page for a particular year is only created once, no matter how many papers were published in that year.

Below is the same query in WebOQL.

```
select unique [Url: x.year, Label:"YearPage"]
              as "RootPage",
              [ label: "Paper" / x ] as x.year
from x in browse("bibtex: myfile.bib")
```

1

```
select [year: y.url] + y as y.url
from y in "browse(RootPage)"
```

The WebOQL query consists of two subqueries, with the web resulting from the first one "piped" into the second one using the `"|"` operator. The first subquery builds the Root, Paper, and Year pages, and the second one redefines each Year page by adding the "year" field to it.

Florid

FLORID [HLLS97, LHL⁺98] is a prototype implementation of the deductive and object-oriented formalism F-logic [KLW95]. To use FLORID as a web query engine, a web document is modeled by the following two classes:

```
url::string[get => webdoc].
```

```
webdoc::string[url => url; author => string;
               modif => string;
               type => string; hrefs@ (string) =>> url;
               error =>> string].
```

The first declaration introduces a class `url`, subclass of `string` with the only method `get`. The notation `get => webdoc` means that `get` is a single-valued method that returns an object of type `webdoc`. The method `get` is system-defined; the effect of invoking `u.get` for a url `u` in the head of a deductive rule is to retrieve from the web the document with that URL and cache it in the local FLORID database as a `webdoc` object with object identifier `u.get`.

The class `webdoc` with methods `self`, `author`, `modif`, `type`, `hrefs` and `error` models the basic information common to all web documents. The notation `hrefs@ (string) =>> url` means that the multi-valued method `hrefs` takes a string as argument and returns a set of objects of type `url`. The idea is that, if `d` is a `webdoc`, then `d.hrefs@ (aLabel)` returns all URL's of documents pointed to by links labeled `aLabel` within document `d`.

Subclasses of documents can be declared as needed using F-logic inheritance, e.g.:

```
htmldoc::webdoc[title => string; text => string].
```

Computation in FLORID is expressed by sets of deductive rules. For example, the program below extracts from the web the set of all documents reachable directly or indirectly from the URL `www.cs.toronto.edu` by links whose labels contain the string "database."

```
("www.cs.toronto.edu":url).get.
(Y:url).get <-
  (X:url).get[hrefs@ (L) =>> {Y}],
  substr("database", L).
```

FLORID provides a powerful formalism for manipulating semi-structured data in a web context. However, it does not currently support the construction of new webs as results of computation; the result is always a set of F-logic objects in the local store.

Ulixes and Penelope

In the ARANEUS project [AMM97b], the query and restructuring process is split into two phases. In the first phase, the UNIXES language is used to build relational views over the web. These views can then be analyzed and integrated using standard database techniques. UNIXES queries extract relational data from instances of page schemes defined in the ADM model, making heavy use of (star-free) path expressions. The second phase consists of generating hypertextual views of the data using the PENELOPE language. Query optimization for relational views over sets of web pages, such as those constructed by UNIXES, is discussed in [MMM98].

Interactive query interfaces

All the languages in the previous two subsections are too complex to be used directly by interactive users, just as SQL is; like SQL, they are meant to be used mostly as programming tools. There has however been work in the design of

interactive query interfaces suitable for casual users. For example, Dataguides [GW97] is an interactive query tool for semistructured databases based on hierarchical summaries of the data graph; extensions to support querying single complex web sites are described in [GW98]. The system described in [HML⁺98] supports queries that combine multimedia features, such as similarity to a given sketch or image, textual features such as keywords, and domain semantics.

Theory of web queries

In defining the semantics of first-generation web query languages, it was immediately observed that certain easily stated queries, such as "list all web documents that no other document points to," could be rather hard to execute. This leads naturally to questions of query computability in this context. Abiteboul and Vianu [AV97a] and Mendelzon and Milo [MM97] propose formal ways of categorizing web queries according to whether they can in principle be computed or not; the key idea being that essentially, the only possible way to access the web is to navigate links from known starting points. (Note this includes a special case navigating links from the large collections of starting points known as index servers or search engines.) Abiteboul and Vianu [AV97b] also discuss fundamental issues posed by query optimization in path traversal queries. Mihaila, Milo and Mendelzon [MMM97] show how to analyze WebSQL queries in terms of the maximum number of web sites. Florescu, Levy and Suciu [FLS98] describe an algorithm for query containment for queries with regular path expressions, which is then used for verifying integrity constraints on the structure of web sites [FFLS98].

4 Information Integration

As stated earlier, the WWW contains a growing number of information sources that can be viewed as *containers* of sets of tuples. These "tuples" can either be embedded in HTML pages, or be hidden behind form interfaces. By writing specialized programs called *wrappers*, one can give the illusion that the web site is serving sets of tuples. We refer to the combination of the underlying web site and the wrapper associated with it as a *web source*.

The task of a web information integration system is to answer queries that may require extracting and combining data from multiple web sources. As an example, consider the domain of movies. The Internet Movie Database contains comprehensive data about movies, their casts, genres and directors. Reviews of movies can be found in multiple other web sources (e.g., web sites of major newspapers), and several web sources provide schedules of movie showings. By combining the data from these sources we can answer queries such as: *give me a movie, playing time and a review of movies starring Frank Sinatra, playing tonight in Paris.*

Several systems have been built with the goal of answering queries using a multitude of web sources [GMPQ⁺97, EW94, WBJ⁺95, LRO96, FW97, DG97b, AKS96, Coh98, AAB⁺98, BEM⁺98]. Many of the problems encountered in building these systems are similar to those addressed in building heterogeneous database systems [ACPS96, WAC⁺93, HZ96, TRV98, FRV96, Bla96, HKWY97]. Web data integration systems have, in addition, to deal with (1) large and evolving number of web sources, (2) little meta-data about the characteristics of the source, and (3) larger degree of source

autonomy.

An important distinction in building data integration systems, and therefore in building web data integration systems, is whether to take a warehousing or a virtual approach (see [HZ96, Hul97] for a comparison). In the warehousing approach, data from multiple web sources is loaded into a warehouse, and all queries are applied to the warehoused data; this requires that the warehouse be updated when data changes, but the advantage is that adequate performance can be guaranteed at query time. In the virtual approach, the data remains in the web sources, and queries to the data integration system are decomposed at run time into queries on the sources. In this approach, data is not replicated, and is guaranteed to be fresh at query time. On the other hand, because the web sources are autonomous, more sophisticated query optimization and execution methods are needed to guarantee adequate performance. The virtual approach is more appropriate for building systems where the number of sources is large, the data is changing frequently, and there is little control over the web sources. For these reasons, most of the recent research has focused on the virtual approach, and therefore, so will our discussion. We emphasize that many of the issues that arise in the virtual approach also arise in the warehousing approach (often in a slightly different form), and hence our discussion is relevant to both cases. Finally, we refer the reader to two commercial applications of web data integration, one that takes the warehousing approach [Jun98] and the other that takes the virtual approach [Jan98].

A prototypical architecture of a virtual data integration system is shown in Figure 3. There are two main features distinguishing such a system from a traditional database system:

- As stated earlier, the system does not communicate directly with a local storage manager. Instead, in order to obtain data, the query execution engine communicates with a set of wrappers. A wrapper is a program which is specific to every web site, and whose task is to translate the data in the web site to a form that can be further processed by the data integration system. For example, the wrapper may extract from an HTML file a set of tuples. It should be emphasized that the wrapper provides only an interface to the data served by the web site, and hence, if the web site provides only limited access to the data (e.g., through a form interface that requires certain inputs), then the wrapper can only support limited access patterns to the data.
- The second difference from traditional systems is that the user does not pose queries directly in the schema in which the data is stored. The reason for this is that one of the principal goals of a data integration system is to free the user from having to know about the specific data sources and interact with each one. Instead, the user poses queries on a *mediated schema*. A mediated schema is a set of *virtual relations*, which are designed for a particular data integration application. The relations in the mediated schema are not actually stored anywhere. As a consequence, the data integration system must first *reformulate* a user query into a query that refers directly to the schemas in the sources. In order to perform the reformulation step, the data integration system requires a set of *source descriptions*. A description of an information source specifies the contents of the source (e.g., contains movies), the at-

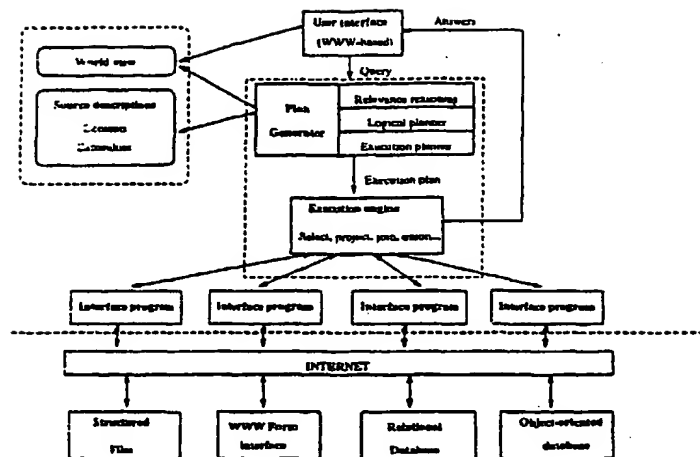


Figure 2: Architecture of a data integration system

tributes that can be found in the source (e.g., genre, cast), constraints on the contents of the source (e.g., contains only American movies), completeness and reliability of the source, and finally, the query processing capabilities of the source (e.g., can perform selections, or can answer arbitrary SQL queries).

The following are the main issues addressed in the work on building web data integration systems.

Specification of mediated schema and reformulation: The mediated schema in a data integration system is the set of collection and attribute names that are used to formulate queries. To evaluate a query, the data integration system must translate the query on the mediated schema into a query on the data sources, that have their own local schemas. In order to do so, the system requires a set of source descriptions. Several recent research works addressed the problem of how to specify source descriptions and how to use them for query reformulation. Broadly speaking, two general approaches have been proposed: *Global as view* (GAV) [GMPQ⁺97, PAGM96, ACPS96, HKWY97, FRV96, TRV98] and *Local as view* (LAV) [LRO96, KW96, DG97a, DG97b, FW97] (see [Ull97] for a detailed comparison).

In the GAV approach, for each relation R in the mediated schema, we write a query over the source relations specifying how to obtain R 's tuples from the sources. The LAV approach takes the opposite approach. For every information source S , we write a query over the relations in the mediated schema that describes which tuples are found in S . The main advantage of the GAV approach is that query reformulation is very simple, because it reduces to view unfolding. In contrast, in the LAV approach it is simpler to add or delete sources because the descriptions of the sources do not have to take into account the possible interactions with other sources, as in the GAV approach, and it is also easier to describe constraints on the contents of sources. The problem of reformulation becomes a variant on the of the problem of answering queries using views [YL87, TSI96, LMSS95, CKPS95, RSU95, DG97b].

Completeness of data in web sources: In general, sources that we find on the WWW are not necessarily complete for

the domain they are covering. For example, a bibliography source is unlikely to be complete for the field of Computer Science. However, in some cases, we can assert completeness statements about sources. For example, the DB&LP Database² has the complete set of papers published in most major database conferences. Knowledge of completeness of a web source can help a data integration system in several ways. Most importantly, since a *negative* answer from a complete source is meaningful, the data integration system can prune access to other sources. The problem of describing completeness of web sources and using this information for query processing is addressed in [Mot89, EGW94, Lev96, Dus97, AD98, FW97]. The work described in [FKL97] describes a probabilistic formalism for describing the contents and overlaps among information sources, and presents algorithms for choosing optimally between sources.

Differing query processing capabilities: From the perspective of the web data integration system, the web sources appear to have vastly differing query processing capabilities. The main reasons for the different appearance are (1) the underlying data may actually be stored in a structured file or legacy system and in this case the interface to this data is naturally limited, and (2) even if the data is stored in a traditional database system, the web site may provide only limited access capabilities for reasons of security or performance.

To build an effective data integration system, these capabilities need to be explicitly described to the system, adhered to, and exploited as much as possible to improve performance. We distinguish two types of capabilities: negative capabilities that limit the access patterns to the data, and positive capabilities, where a source is able to perform additional algebraic operations in addition to simple data fetches.

The main form of negative capabilities is limitations on the binding patterns that can be used in queries sent to the source. For example, it is not possible to send a query to the Internet Movie Database asking for *all* the movies in the database and their casts. Instead, it is only possible to ask for the cast of *given* movie, or to ask for the set of movies in which a particular actor appears. Several works have con-

²<http://www.informatik.uni-trier.de/~ley/db/>

sidered the problem of answering queries in the presence of binding pattern limitations [RSU95, KW96, LRO96, FW97].

Positive capabilities pose another challenge to a data integration system. If a data source has the ability to perform operations such as selections and joins, we would like to push as much as possible of the processing to the source, thereby hopefully reducing the amount of local processing and the amount of data transmitted over the network. The problem of describing the computing capabilities of data sources and exploiting them to create query execution plans is considered in [PGGMU95, TRV98, LRU96, HKWY97, VP97a].

Query optimization: Many works on web data integration systems have focused on the problem of selecting a *minimal* set of web sources to access, and on determining the minimal query that needs to be sent to each one. However, the issue of choosing an optimal query execution plan to access the web sources has received relatively little attention in the data integration literature [HKWY97], and remains an active area of research. The additional challenges that are faced in query optimization over sources on the WWW is that we have few statistics on the data in the sources, and hence little information to evaluate the cost of query execution plans. The work in [NGT98] considers the problem of calibrating the cost model for query execution plans in this context. The work in [YPAGM98] discusses the problem of query optimization for *fusion queries*, which are a special class of integration queries that focus on retrieving various attributes of a given object from multiple sources. Moreover, we believe that query processing in data integration systems is one area which would benefit from ideas such as interleaving of planning and execution and of computing conditional plans [GC94, KD98].

Query execution engines: Even less attention has been paid to the problem of building query execution engines targeted for web data integration. The challenges in building such engines are caused by the autonomy of the data sources and the unpredictability of the performance of the network. In particular, when accessing web sources we may experience initial delays before data is transmitted, and even when it is, the arrival of the data may be bursty. The work described in [AFT98, UFA98] has considered the problem of adapting a query execution plans to initial delays in the arrival of the data.

Wrapper construction: Recall that the role of a wrapper is to extract the data out of a web site into a form that can be manipulated by the data integration system. For example, the task of a wrapper could be to pose a query to a web source using a form interface, and to extract a set of answer tuples out of the resulting HTML page. The difficulty in building wrappers is that the HTML page is usually designed for human viewing, rather than for extracting data by programs. Hence, the data is often embedded in natural language text or hidden within graphical presentation primitives. Moreover, the form of the HTML pages changes frequently, making it hard to maintain the wrappers. Several works have considered the problem of building tools for rapid creation of wrappers. One class of tools (e.g., [HGMN⁺98, GRVB98]) is based on developing specialized grammars for specifying how the data is laid out in an HTML page, and therefore how to extract the required data. A second class of techniques is based on developing inductive learning techniques for automatically learning a wrapper. Using these algorithms, we provide the system

with a set of HTML pages where the data in the page is *labeled*. The algorithm uses the labeled examples to automatically output a grammar by which the data can be extracted from subsequent pages. Naturally, the more examples we give the system, the more accurate the resulting grammar can be, and the challenge is to discover wrapper languages that can be learned with a small number of examples. The first formulation of wrapper construction as inductive learning and a set of algorithms for learning simple classes of wrappers are given in [KDW97]. The algorithm described in [AK97] exploits heuristics specific to the common uses of HTML in order to obtain faster learning. It should be noted that Machine Learning methods have also been used to learn the mapping between the source schemas and the mediated schemas [PE95, DEW97]. The work described [CDF⁺98] is a first step in bridging the gap between the approaches of Machine Learning and of Natural Language Processing to the problem of wrapper construction. Finally, we note that the emergence of XML may lead web site builders to export the data underlying their sites in a machine readable form, thereby greatly simplifying the construction of wrappers.

Matching objects across sources: One of the hardest problems in answering queries over a multitude of sources is deciding that two objects mentioned in two different sources refer to the same entity in the world. This problem arises because each source employs its own naming conventions and shorthands. Most systems deal with this problem using domain specific heuristics (as in [FHM94]). In the WHIRL system [Coh98], matching of objects across sources is done by using techniques from Information Retrieval. Furthermore, the matching of the objects is elegantly integrated in a novel query execution algorithm.

5 Web site construction and restructuring

The previous two sections discussed tasks that concerned querying *existing* web sites and their content. However, given the fact that web sites essentially provide access to complex structures of information: it is natural to apply techniques from Database Systems to the process of *building* and *maintaining* web sites. One can distinguish two general classes of web site building tasks: one in which web sites are created from a collection of underlying data sources, and another in which they are created by restructuring existing web sites. As it turns out, the same techniques are required for both of these classes. Furthermore, we note that the task of providing a web interface to data that exists in a single database system [NS96] is a simple instance of the problem of creating web sites.³

To understand the problem of building web sites and the possible import of database technology, consider the tasks that a web site builder must address: (1) choosing and accessing the data that will be displayed at the site, (2) designing the site's structure, i.e., specifying the data contained within each page and the links between pages, and (3) designing the graphical presentation of pages. In existing web site management tools, these tasks are, for the most part, interdependent. Without any site-creation tools, a site builder writes HTML files by hand or writes programs to produce them and must focus simultaneously on a page's content, its relationship to other pages, and its graphical presentation. As a result, several important tasks, such as automatically

³Most database vendors were quick to provide commercial tools for performing this task.

- updating a site, restructuring a site, or enforcing integrity constraints on a site's structure: are tedious to perform.

Web sites as declaratively defined structures: Several systems have been developed with the goal of applying database techniques to the problem of web site creation [FFK⁺98, AMM98, AM98, CDSS98, PF98, JB97, LSB⁺98, TN98]. The common theme to these systems is that they provide an explicit declarative representation of the structure of a web site. The structure of the web site is defined as a *view* over existing data. However, we emphasize that the languages used to create these views result in *graphs* of web pages with hypertext links, rather than simple tables. The systems differ on the data model they use, the query language they use, and whether they have an intermediate logical representation of the web site, rather than having only a representation of the final HTML.

Building a web site using a declarative representation of the structure of the site has several significant advantages. Since a web site's structure and content are defined declaratively by a query, not procedurally by a program, it is easy to create multiple *versions* of a site. For example, it is possible to easily build internal and external views of an organization's site or to build sites tailored to different classes of users. Currently, creating multiple versions requires writing multiple sets of programs or manually creating different sets of HTML files. Building multiple versions of a site can be done by either writing different site definition queries, or by changing the graphical representation independently of the underlying structure. Furthermore, a declarative representation of the web site's structure also supports easy evolution of a web site's structure. For example, to reorganize pages based on frequent usage patterns [PE97], or to extend the site's content, simply rewrite the site-definition query, as opposed to rewriting a set of programs or a set of HTML files. Declarative specification of web sites can offer other advantages. For example, it becomes possible to express and enforce integrity constraints on the site [FFLS98], and to update a site incrementally when changes occur in the underlying data. Moreover, a declarative specification provides a platform for developing optimization algorithms for run-time management of data intensive web sites. The challenge in run-time management of a web site is to automatically find an optimal tradeoff between precomputation of parts of the web site and click-time computation. Finally, we remark that building web sites using this paradigm will also facilitate the tasks of querying web sites and integrating data from multiple web sources.

A prototypical architecture of such a system is shown in Figure 3. At the bottom level, the system accesses a set of data sources containing the data that will be served on the web site. The data may be stored in databases, in structured files, or in existing web sites. The data is represented in the system in some data model, and the system provides a 'uniform' interface to these data sources using techniques similar to the ones described in the previous section. The main step in building a web site is to write an expression that declaratively represents the structure of the web site. The expression is written in a specific query language provided by the system. The result of applying this query to the underlying data is the logical representation of the web site in the data model of the system (e.g., a labeled directed graph). Finally, to actually create a browsable web site, the system contains a method (e.g., HTML templates) for translating the logical structure into a set of HTML files.

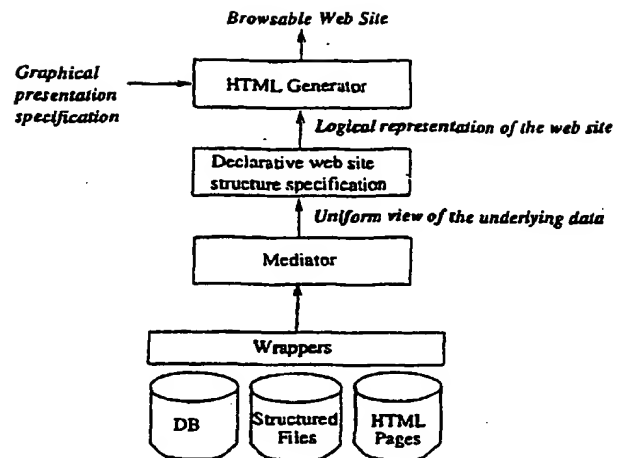


Figure 3: Architecture for Web Site Management Systems

Some of the salient characteristics of the different systems are the following. STRUDEL [FFK⁺98] uses a semistructured data model of labeled directed graphs for modeling both the underlying data and for modeling the web site. It uses a single query language, STRUQL, throughout the system, both for integrating the raw data and for defining the structure of the web site. ARANEUS [AMM97b] uses a more structured data model, ADM, and provides a language for transforming data into ADM and a language for creating web sites from data modeled in ADM. In addition, Araneus uses web-specific constructs in the data model. The Autoweb [PF98] system is based on the hypermedia design model (HDM), a design tool for hypermedia applications. Its data model is based on the entity-relationship model; the "access schema" specifies how the hyperbase is navigated and accessed in a browsable site; and the "presentation schema" specifies how objects and paths in the hyperbase and access schemas are rendered. All three systems mentioned above provide a clear separation between the creation of the logical structure of the web site and the specification of the graphical presentation of the site. The YAT system [CDSS98] is an application of a data conversion language to the problem of building web sites. Using YAT, a web site designer writes a set of rules converting from the raw data into an abstract syntax tree of the resulting HTML, without going through an intermediate logical representation phase. In fact, in a similar way, other languages for data conversion (such as [MZ98, MPP⁺93, PMSL94]) can also be used to build web sites. The WIRM system [JB97] is similar in spirit to the above systems in that it enables users to build web sites in which the pages can be viewed context-sensitive views of the underlying data. The major focus of WIRM is on integrating medical research data for the national Human Brain Project.

6 Conclusions, Perspectives and Future Work

An overarching question regarding the topic of this survey is whether the World-Wide Web presents novel problems to the database community. In many ways, the WWW is not similar to a database. For example, there is no uniform structure, no integrity constraints, no transactions, no

standard query language or data model. And yet, as the survey has showed, the powerful abstractions developed in the database community may prove to be key in taming the web's complexity and providing valuable services.

Of particular importance is the view of a large web site as being not just a database, but an information system built around one or more databases with an accompanying complex navigation structure. In that view, a web site has many similarities to non-web information systems. Designing such a web site requires extending information systems design methodologies [AMM98, PF98]. Using these principles to build web sites will also impact the way we query the web and the way we integrate data from multiple web sources.

Several trends will have significant impact on the use of database technology for web applications. The first is, of course, XML. The considerable momentum behind XML and related metadata initiatives can only help the applicability of database concepts to the web by providing the much needed structure in a widely accepted format. While the availability of data in XML format will reduce the need to focus on wrappers converting human readable data to machine readable data, the challenges of semantic integration of data from web sources still remains. Building on our experience in developing methods for manipulating semistructured data, our community is in a unique position to develop tools for manipulating data in XML format. In fact, some of the concepts developed in this community are already being adapted to the XML context [DFF⁺98, GMW98]. Other projects under way in the database community in the area of metadata architectures and languages (e.g. [MRT98, KMSS98]) are likely to take advantage of and merge with the XML framework.

A second trend that will affect the applicability of database techniques for querying the web is the growth of the so-called *hidden web*. The hidden web refers to the web pages that are generated by programs given user inputs, and are therefore not accessible to web crawlers for indexing. A recent article [LG98] claims that close to 80% of the web is already in the hidden web. If our tools are to be able to benefit from data in the hidden web, we must develop techniques for identifying sites that generate web pages, classify them and automatically create query interfaces to them.

There is no shortage in possible directions for future research in this area. In the past, the bulk of the work has focused on the logical level, developing appropriate data models, query languages and methods for describing different aspects of web sources. In contrast, problems of query optimization and query execution have received relatively little attention in the database community, and pose some of the more important challenges for future work. Some of the important directions in which to enrich our data models and query languages include the incorporation of various forms of meta data about sources (e.g., probabilistic information) and the principled combination of querying structured and unstructured data sources on the WWW.

Finally, in this article we tried to provide a representative list of references on the topic of web and databases. In addition to these references, readers can get a more detailed account from recent workshops related to the topic of the survey [SSD97, Web98, AI198].

Acknowledgements

The authors are grateful to the following colleagues who provided suggestions and comments on earlier versions of this paper: Serge Abiteboul, Gustavo Arocena, Paolo Atzeni, José Blakeley, Nick Kushmerick, Bertram Ludäscher, C. Mohan, Yannis Papakonstantinou, Oded Shmueli, Anthony Tomasic and Dan Weld.

References

- [AAB⁺98] Jos Luis Ambite, Naveen Ashish, Greg Barish, Craig A. Knoblock, Steven Minton, Pragnesh J. Modi, Ion Muslea, Andrew Philpot, and Sheila Tejada. ARIADNE: A system for constructing mediators for internet sources (system demonstration). In *Proc. of ACM SIGMOD Conf. on Management of Data*, Seattle, WA, 1998.
- [Abi97] Serge Abiteboul. Querying semi-structured data. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.
- [ACM93] Serge Abiteboul, Sophie Cluet, and Tova Milo. Querying and updating the file. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Dublin, Ireland, 1993.
- [ACPS96] S. Adali, K. Candan, Y. Papakonstantinou, and V.S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Montreal, Canada, 1996.
- [AD98] S. Abiteboul and O. Duschka. Complexity of answering queries using materialized views. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Seattle, WA, 1998.
- [AFT98] Laurent Amsaleg, Michael Franklin, and Anthony Tomasic. Dynamic query operator scheduling for wide-area remote access. *Distributed and Parallel Databases*, 6(3):217-246, July 1998.
- [AI198] *Proceedings of the AAAI Workshop on Intelligent Data Integration*, Madison, Wisconsin, July 1998.
- [AK97] Naveen Ashish and Craig A. Knoblock. Wrapper generation for semi-structured internet sources. *SIGMOD Record*, 26(4):8-15, 1997.
- [AKS96] Yigal Arens, Craig A. Knoblock, and Wei-Min Shen. Query reformulation for dynamic information integration. *International Journal on Intelligent and Cooperative Information Systems*, (6) 2/3:99-130, June 1996.
- [AM98] Gustavo Arocena and Alberto Mendelzon. WebOQL: Restructuring documents, databases and webs. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, Orlando, Florida, 1998.
- [AMM97a] Gustavo O. Arocena, Alberto O. Mendelzon, and George A. Mihaila. Applications of a web query language. In *Proc. of the Int. WWW Conf.*, April 1997.

- [AMM97b] Paolo Atzeni, Giansalvatore Mecca, and Paolo Meriardo. To weave the web. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 1997.
- [AMM98] Paolo Atzeni, Giansalvatore Mecca, and Paolo Meriardo. Design and maintenance of data-intensive web sites. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, Valencia, Spain, 1998.
- [AMR⁺98] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet Weiner. Incremental maintenance for materialized views over semistructured data. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, New York City, USA, 1998.
- [AQM⁺97] Serge Abiteboul, Dallen Quass, Jason McHugh, Jennifer Widom, and Janet Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68-88, April 1997.
- [Aro97] Gustavo Arocena. WebOQL: Exploiting document structure in web queries. Master's thesis, University of Toronto, 1997.
- [AV97a] S. Abiteboul and V. Vianu. Queries and computation on the Web. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.
- [AV97b] Serge Abiteboul and Victor Vianu. Regular path queries with constraints. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Tucson, Arizona, May 1997.
- [BBH⁺98] Krishna Bharat, Andrei Broder, Monika Henzinger, Puneet Kumar, and Suresh Venkatasubramanian. The connectivity server: fast access to linkage information on the web. In *Proc. of the Int. WWW Conf.*, April 1998.
- [BBMR89] Alex Borgida, Ronald Brachman, Deborah McGuinness, and Lori Resnick. CLASSIC: A structural data model for objects. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 59-67, Portland, Oregon, 1989.
- [BDFS97] Peter Buneman, Susan Davidson, Mary Fernandez, and Dan Suciu. Adding structure to unstructured data. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.
- [BDH⁺95] C. M. Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. The harvest information discovery and access system. *Computer Networks and ISDN Systems*, 28(1-2):119-125, December 1995.
- [BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 505-516, Montreal, Canada, 1996.
- [BEM⁺98] C. Beeri, G. Elber, T. Milo, Y. Sagiv, O. Shmueli, N. Tishby, Y. Kogan, D. Konopnicki, P. Mogilevski, and N. Slonim. Websuite-a tool suite for harnessing web data. In *Proceedings of the International Workshop on the Web and Databases*, Valencia, Spain, 1998.
- [BH98] Krishna Bharat and Monika Henzinger. Improved algorithms for topic distillation in hyperlinked environments. In *Proc. 21st Int'l ACM SIGIR Conference*, 1998.
- [Bil98] David Billard. Transactional services for the web. In *Proceedings of the International Workshop on the Web and Databases*, pages 11-17, Valencia, Spain, 1998.
- [BK90] C. Beeri and Y. Kornatzky. A logical query language for hypertext systems. In *Proc. of the European Conference on Hypertext*, pages 67-80. Cambridge University Press, 1990.
- [Bla96] Jose A. Blakeley. Data access for the masses through OLE DB. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 161-172, Montreal, Canada, 1996.
- [BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. of the Int. WWW Conf.*, April 1998.
- [BT98] Philippe Bonnet and Anthony Tomasic. Partial answers for unavailable data sources. In *Proceedings of the 1998 Workshop on Flexible Query-Answering Systems (FQAS'98)*, pages 43-54. Department of Computer Science, Roskilde University, 1998.
- [Bun97] Peter Buneman. Semistructured data. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 117-121, Tucson, Arizona, 1997.
- [CAC94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 313-324, Minneapolis, Minnesota, 1994.
- [CDF⁺98] Mark Craven, Dan DiPasquo, Dayne Freitag, Andrew McCallum, Tom Mitchell, Kamal Nigam, and Sean Slattery. Learning to extract symbolic knowledge from the world-wide web. In *Proceedings of the AAAI Fifteenth National Conference on Artificial Intelligence*, 1998.
- [CDRR98] Soumen Chakrabarti, Byron Dom, Prabhakar Raghavan, and Sridhar Rajagopalan. Automatic resource compilation by analyzing hyperlink structure and associated text. In *Proc. of the Int. WWW Conf.*, April 1998.
- [CDSS98] Sophie Cluet, Claude Delobel, Jerome Simeon, and Katarzyna Smaga. Your mediators need data conversion. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Seattle, WA, 1998.
- [CGL98] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. What can knowledge representation do for semi-structured data? In *Proceedings of the AAAI Fifteenth National Conference on Artificial Intelligence*, 1998.

- [CGMP98] Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. Efficient crawling through url ordering. In *Proc. of the Int. WWW Conf.*, April 1998.
- [CK98] J. Carriere and R. Kazman. Webquery: Searching and visualizing the web through connectivity. In *Proc. of the Int. WWW Conf.*, April 1998.
- [CKPS95] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, Taipei, Taiwan, 1995.
- [CM89] Mariano P. Consens and Alberto O. Mendelzon. Expressing structural hypertext queries in graphlog. In *Proc. 2nd. ACM Conference on Hypertext*, pages 269-292, Pittsburgh, November 1989.
- [CM90] Mariano Consens and Alberto Mendelzon. GraphLog: a visual formalism for real life recursion. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 404-416, Atlantic City, NJ, 1990.
- [CMW87] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 323-330, San Francisco, CA, 1987.
- [CMW88] I.F. Cruz, A.O. Mendelzon, and P.T. Wood. G+: Recursive queries without recursion. In *Proceedings of the Second International Conference on Expert Database Systems*, pages 355-368, 1988.
- [Coh98] William Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Seattle, WA, 1998.
- [CS98] Pei Cao and Sekhar Sarukkai, editors. *Proceedings of Workshop on Internet Server Performance*, Madison, Wisconsin, 1998. <http://www.cs.wisc.edu/cao/WISP98-program.html>.
- [DEW97] B. Doorenbos, O. Etzioni, and D. Weld. Scalable comparison-shopping agent for the world-wide web. In *Proceedings of the International Conference on Autonomous Agents*, February 1997.
- [DFF⁺98] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. <http://www.research.att.com/~mfj/xml/w3c-note.html>, 1998.
- [DG97a] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Tucson, Arizona, 1997.
- [DG97b] Oliver M. Duschka and Michael R. Genesereth. Query planning in infomaster. In *Proceedings of the ACM Symposium on Applied Computing*, San Jose, CA, 1997.
- [Dus97] Oliver Duschka. Query optimization using local completeness. In *Proceedings of the AAAI Fourteenth National Conference on Artificial Intelligence*, 1997.
- [EGW94] Oren Etzioni, Keith Golden, and Daniel Weld. Tractable closed world reasoning with updates. In *Proceedings of the Conference on Principles of Knowledge Representation and Reasoning, KR-94*, 1994. Extended version to appear in *Artificial Intelligence*.
- [EW94] Oren Etzioni and Dan Weld. A softbot-based interface to the internet. *CACM*, 37(7):72-76, 1994.
- [FFK⁺98] Mary Fernandez, Daniela Florescu, Jaewoo Kang, Alon Levy, and Dan Suciu. Catching the boat with Strudel: Experiences with a web-site management system. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Seattle, WA, 1998.
- [FFLS97] Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for a web-site management system. *SIGMOD Record*, 26(3):4-11, September 1997.
- [FFLS98] Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. Reasoning about web-sites. In *Working notes of the AAAI-98 Workshop on Artificial Intelligence and Data Integration*. American Association of Artificial Intelligence, 1998.
- [FHM94] Douglas Fang, Joachim Hammer, and Dennis McLeod. The identification and resolution of semantic heterogeneity in multidatabase systems. In *Multidatabase Systems: An Advanced Solution for Global Information Sharing*, pages 52-60. IEEE Computer Society Press, Los Alamitos, California, 1994.
- [FKL97] Daniela Florescu, Daphne Koller, and Alon Levy. Using probabilistic information in data integration. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 216-225, Athens, Greece, 1997.
- [FLS98] Daniela Florescu, Alon Levy, and Dan Suciu. Query containment for conjunctive queries with regular expressions. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Seattle, WA, 1998.
- [FRV96] Daniela Florescu, Louisa Raschid, and Patrick Valduriez. A methodology for query reformulation in cis using semantic knowledge. *Int. Journal of Intelligent & Cooperative Information Systems, special issue on Formal Methods in Cooperative Information Systems*, 5(4), 1996.
- [FW97] M. Friedman and D. Weld. Efficient execution of information gathering plans. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Nagoya, Japan, 1997.

- [GC94] G. Graefe and R. Cole. Optimization of dynamic query evaluation plans. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Minneapolis, Minnesota, 1994.
- [GGMT99] Luis Gravano, Hector Garcia-Molina, and Anthony Tomasic. GIOSS: Text-source discovery over the internet. *ACM Transactions on Database Systems (to appear)*, 1999.
- [GMPQ⁺97] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogenous information sources, March 1997.
- [GMW98] R. Goldman, J. McHugh, and J. Widom. Lore: A database management system for XML. Presentation. Stanford University Database Group, 1998.
- [GRC97] S. Gadde, M. Rabinovich, and J. Chase. Reduce, reuse, recycle: An approach to building large internet caches. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, May 1997.
- [GRVB98] Jean-Robert Gruser, Louiqa Raschid, Maria Esther Vidal, and Laura Bright. Wrapper generation for web accessible data sources. In *Proceedings of the CoopIS*, 1998.
- [GW97] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Athens, Greece, 1997.
- [GW98] Roy Goldman and Jennifer Widom. Interactive query and search in semistructured databases. In *Proceedings of the International Workshop on the Web and Databases*, pages 42-48, Valencia, Spain, 1998.
- [GZC89] Ralf Hartmut Güting, Roberto Zicari, and David M. Choy. An algebra for structured office documents. *ACM TOIS*, 7(2):123-157, 1989.
- [Hal88] Frank G. Halasz. Reflections on Notecards: Seven issues for the next generation of hypermedia systems. *Comm. of the ACM*, 31(7):836-852, 1988.
- [Har94] Coleman Harrison. Aql: An adaptive query language. Technical Report NU-CCS-94-19, Northeastern University, October 1994. Master's Thesis.
- [HGMN⁺98] Joachim Hammer, Hector Garcia-Molina, Svetlozar Nestorov, Ramana Yerneni, Markus M. Breunig, and Vasilis Vassalos. Template-based wrappers in the TSIMMIS system (system demonstration). In *Proc. of ACM SIGMOD Conf. on Management of Data*, Tucson, Arizona, 1998.
- [HKWY97] Laura Haas, Donald Kossmann, Edward Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Athens, Greece, 1997.
- [HLLS97] Rainer Himmeröder, Georg Lausen, Bertram Ludäscher, and Christian Schlepphorst. On a declarative semantics for web queries. In *Proc. of the Int. Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 386-398, Singapore, December 1997. Springer.
- [HML⁺98] Kyoji Hirata, Sougata Mukherjea, Wen-Syan Li, Yusaku Okamura, and Yoshinori Hara. Facilitating object-based navigation through multimedia web databases. *TAPOS*, 4(4), 1998. In press.
- [Hul97] Richard Hull. Managing semantic heterogeneity in databases: A theoretical perspective. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 51-61, Tucson, Arizona, 1997.
- [HZ96] Richard Hull and Gang Zhou. A framework for supporting data integration using the materialized and virtual approaches. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 481-492, Montreal, Canada, 1996.
- [Jan98] <http://www.jango.excite.com>. 1998.
- [JB97] R. Jakobovits and J. F. Brinkley. Managing medical research data with a web-interfacing repository manager. In *American Medical Informatics Association Fall Symposium*, pages 454-458, Nashville, Oct 1997.
- [Jun98] <http://www.junglee.com>, 1998.
- [KD98] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 106-117, Seattle, WA, 1998.
- [KDW97] N. Kushmerick, R. Doorenbos, and D. Weld. Wrapper induction for information extraction. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, 1997.
- [Kle98] Jon Kleinberg. Authoritative sources in a hyperlinked environment. In *Proceedings of 9th ACM-SIAM Symposium on Discrete Algorithms*, 1998.
- [KLW95] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *J. ACM*, 42(4):741-843, 1995.
- [KMSS98] Y. Kogan, D. Michaeli, Y. Sagiv, and O. Shmueli. Utilizing the multiple facets of www contents. *Data and Knowledge Engineering*, 1998. In press.
- [KS95] D. Konopnicki and O. Shmueli. W3QS: A query system for the World Wide Web. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 54-65, Zurich, Switzerland, 1995.
- [KS98] David Konopnicki and Oded Shmueli. Bringing database functionality to the WWW. In *Proceedings of the International Workshop on the Web and Databases*, Valencia, Spain, pages 49-55, 1998.

- [KW96] Chung T. Kwok and Daniel S. Weld. Planning to gather information. In *Proceedings of the AAAI Thirteenth National Conference on Artificial Intelligence*, 1996.
- [Lev96] Alon Y. Levy. Obtaining complete answers from incomplete databases. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Bombay, India, 1996.
- [LG98] S. Lawrence and C.L. Giles. Searching the world wide web. *Science*, 280(4):98-100, 1998.
- [LHL⁺98] Bertram Ludäscher, Rainer Himmeröder, Georg Lausen, Wolfgang May, and Christian Schlep-phorst. Managing semistructured data with *FLORID*: A deductive object-oriented perspective. *Information Systems*, 23(8), 1998. to appear.
- [LMSS95] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, San Jose, CA, 1995.
- [LRO96] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Bombay, India, 1996.
- [LRU96] Alon Y. Levy, Anand Rajaraman, and Jeffrey D. Ullman. Answering queries using limited external processors. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Montreal, Canada, 1996.
- [LSB⁺98] T. Lee, L. Sheng, T. Bozkaya, N.H. Balkir, Z.M. Ozsoyoglu, and G. Ozsoyoglu. Querying multimedia presentations based on content. to appear in *IEEE Trans. on Know. and Data Engineering*, 1998.
- [LSCH98] Wen-Syan Li, Junho Shim, K. Selcuk Canadian, and Yoshinori Hara. WebDB: A web query system and its modeling, language, and implementation. In *Proc. IEEE Advances in Digital Libraries '98*, 1998.
- [LSS96] Laks V. S. Lakshmanan, Fereidoon Sadri, and Iyer N. Subramanian. A declarative language for querying and restructuring the Web. In *Proc. of 6th. International Workshop on Research Issues in Data Engineering, RIDE '96*, New Orleans, February 1996.
- [MM97] Alberto O. Mendelzon and Tova Milo. Formal models of web queries. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 134-143. Tucson, Arizona, May 1997.
- [MMM97] A. Mendelzon, G. Mihaila, and T. Milo. Querying the world wide web. *International Journal on Digital Libraries*, 1(1):54-67, April 1997.
- [MMM98] Giansalvatore Mecca, Alberto O. Mendelzon, and Paolo Merialdo. Efficient queries over web views. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, Valencia, Spain, 1998.
- [Mot89] Amihai Motro. Integrity = validity + completeness. *ACM Transactions on Database Systems*, 14(4):480-502, December 1989.
- [MP96] Kenneth Moore and Michelle Peterson. A groupware benchmark based on lotus notes. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, 1996.
- [MPP⁺93] Bernhard Mitschang, Hamid Pirahesh, Peter Pistor, Bruce G. Lindsay, and Norbert Sdkamp. Sql/xnf - processing composite objects as abstractions over relational data. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, pages 272-282. Vienna, Austria, 1993.
- [MRT98] George A. Mihaila, Louiqa Raschid, and Anthony Tomasic. Equal time for data on the internet with websemantics. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, Valencia, Spain, 1998.
- [MZ98] Tova Milo and Sagit Zohar. Using schema matching to simplify heterogeneous data translation. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, New York City, USA, 1998.
- [NAM98] Svetlozar Nestorov, Serge Abiteboul, and Rajeev Motwani. Extracting schema from semistructured data. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Seattle, WA, 1998.
- [NGT98] Hubert Naacke, Georges Gardarin, and Anthony Tomasic. Leveraging mediator cost models with heterogeneous data sources. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, Orlando, Florida, 1998.
- [NS96] Tam Nguyen and V. Srinivasan. Accessing relational databases from the world wide web. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Montreal, Canada, 1996.
- [PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Bombay, India, 1996.
- [PdBA⁺92] Jan Paredaens, Jan Van den Bussche, Marc Andries, Marc Gemis and Marc Gyssens, Inge Thyssens, Dirk Van Gucht, Vijay Sarathy, and Lawrence V. Saxton. An overview of GOOD. *SIGMOD Record*, 21(1):25-31, 1992.
- [PE95] Mike Perkowitz and Oren Etzioni. Category translation: Learning to understand information on the internet. In *Working Notes of the AAAI Spring Symposium on Information Gathering from Heterogeneous Distributed Environments*. American Association for Artificial Intelligence., 1995.

- [PE97] Mike Perkowitz and Oren Etzioni. Adaptive web sites: an AI challenge. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, 1997.
- [PF98] P. Paolini and P. Fraternali. A conceptual model and a tool environment for developing more scalable, dynamic, and customizable web applications. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, Valencia, Spain, 1998.
- [PGGMU95] Yannis Papakonstantinou, Ashish Gupta, Hector Garcia-Molina, and Jeffrey Ullman. A query translation scheme for rapid implementation of wrappers. In *Proc. of the Int. Conf. on Deductive and Object-Oriented Databases (DOOD)*, 1995.
- [PGMW95] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, pages 251-260, Taipei, Taiwan, 1995.
- [PMSL94] Hamid Pirahesh, Bernhard Mitschang, Norbert Sdkamp, and Bruce G. Lindsay. Composite-object views in relational dbms: an implementation perspective. *Information Systems*, 15(1):69-88, 1994.
- [PPR96] P. Pirolli, J. Pitkow, and R. Rao. Silk from a sow's ear: Extracting usable structures from the web. In *Proc. ACM SIGCHI Conference*, 1996.
- [RSU95] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, San Jose, CA, 1995.
- [Spe97] Ellen Spertus. ParaSite: Mining structural information on the web. In *Proc. of the Int. WWW Conf.*, April 1997.
- [SSD97] *Proceedings of Workshop on Management of Semistructured Data*, Tucson, Arizona, May 1997.
- [TMD92] J. Thierry-Mieg and R. Durbin. A C.elegans database: syntactic definitions for the ACEDB data base manager, 1992.
- [TN98] Motomichi Toyama and T. Nagafuji. Dynamic and structured presentation of database contents on the web. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, Valencia, Spain, 1998.
- [TRV98] A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to distributed heterogeneous data sources with Disco. *IEEE Transactions On Knowledge and Data Engineering (to appear)*, 1998.
- [TSI96] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: A versatile tool for physical data independence. *VLDB Journal*, 5(2):101-118, 1996.
- [UFA98] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost based query scrambling for initial delays. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 130-141, Seattle, WA, 1998.
- [Ull97] Jeffrey D. Ullman. Information integration using logical views. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.
- [VP97a] Vasilis Vassalos and Yannis Papakonstantinou. Describing and using the query capabilities of heterogeneous sources. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Athens, Greece, 1997.
- [VP97b] Anne-Marie Vercoustre and François Paradis. A descriptive language for information object reuse through virtual documents. In *Proceedings of the 4th International Conference on Object-Oriented Information Systems (OOIS'97)*, Brisbane, Australia, November 1997.
- [WAC+93] Darrel Woelk, Paul Attie, Phil Cannata, Greg Meredith, Amit Seth, Munindar Sing, and Christine Tomlinson. Task scheduling using intertask dependencies in Carnot. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 491-494, 1993.
- [WBJ+95] Darrell Woelk, Bill Bohrer, Nigel Jacobs, K. Ong, Christine Tomlinson, and C. Unnikrishnan. Carnot and infosleuth: Database technology and the world wide web. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 443-444, San Jose, CA, 1995.
- [Web98] *Proceedings of the International Workshop on the Web and Databases*, Valencia, Spain, March 1998.
<http://poincare.dia.uniroma3.it:8080/webdb98/papers.t>
- [WWW98] *Proceedings of 3d WWW Caching Workshop (Manchester, England)*, June 1998.
- [XML98] <http://w3c.org/XML/>, 1998.
- [YL87] H. Z. Yang and P. A. Larson. Query transformation for PSJ-queries. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 245-254, Brighton, England, 1987.
- [YPAGM98] Ramana Yerneni, Yannis Papakonstantinou, Serge Abiteboul, and Hector Garcia-Molina. Fusion queries over internet databases. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, pages 57-71, Valencia, Spain, 1998.
- [ZGM98] Yue Zhuge and Hector Garcia-Molina. Graph structured views and their incremental maintenance. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, Orlando, Florida, February 1998.